Master's Thesis

Immersive Mesh Creator

Lukas Fraser

Institute of Computer Graphics and Knowledge Visualization, TU Graz

www.cgv.tugraz.at

Abstract

A wide range of devices is nowadays capable of displaying 3D computer graphics. Devices as small as phones have the ability to render rather impressive scenes. PCs can be connected to multiple large flatscreens, allowing a great field of view compared to the small screen of a phone. Special monitors can display 3D graphics with the help of techniques such as stereoscopy. However, normal stereoscopy requires the user to wear special glasses. An improved method called autostereoscopy tries to achieve the same effect, only without the glasses. The picture will therefore no longer appear to be flat. Alternatively, the glasses can be turned into a head-mounted display and no longer require a separate monitor. All these technologies however have one flaw: the user's freedom of movement is very limited. It is almost mandatory to remain at a certain position. If the user tries to look around, special handling is required. With the help of a tracking system, only comparatively small movements with the head can be registered. Movements that go beyond a certain distance are difficult to handle. In addition to the distance, the viewing angle is very limited as well. Depending on the set-up, the head can usually only be tilted slightly.

It is impossible to look at an object from a completely different angle, for example, from the back. Hence, an alternative way of moving around in the scene has to be implemented. An obvious choice are mouse and keyboard, as almost every computer has access to them. Computer gamers are familiar with this kind of interacting with a virtual scene. A key has to be pressed and the character of the player starts to move in one direction. The mouse simulates the movements of the head and can be used to look around. Despite being a very precise and efficient way of navigating in 3D space, it lacks a certain degree of realism. The gamer is sitting still most of the time, only the hands are moving. Therefore it becomes very difficult to *immerse* into the 3D scene. The Institute of Computer Graphics and Knowledge Visualization at the TU Graz has the means to provide a fully immersive 3D experience. DAVE, a virtual environment using multiple projectors and an optical tracking system, enables the user to truly move around in a virtual scene. It is possible to look at an object from different angles by just walking a few steps. This results in a whole new way of interacting with the scene. Many different applications can benefit from these capabilities. However, there also are certain limitations such as input precision or the fact that the user is standing in the middle of an empty room and does not have access to complicated input devices.

This thesis tries to deal with these and other limitations and describes the design and implementation of an application that can be used to create meshes in such a virtual environment. By utilising the capabilities of the DAVE, a user is supposed to feel fully immersed, hence the name *Immersive Mesh Creator*. A selection of tools are developed, enabling the user to quickly draw and sculpt objects in a very intuitive way.

Abstract in German

Heutzutage gibt es eine Vielzahl von Geräten, die 3D Computergrafik darstellen kann. Bereits Geräte von der Größe eines Mobiltelefons besitzen die Fähigkeit, eindrucksvolle Szenen wiederzugeben. PCs wiederum können mit mehreren Flachbildschirmen gleichzeitig verbunden werden und man erhält so eine vielfach größere Bildfläche im Vergleich zu einem kleinen Telefon. Mithilfe der Stereoskopie können Monitore ein Bild dreidimensional darstellen. Hierfür werden zwei getrennte Bilder erstellt, eines für jedes Auge. Überlicherweise wird eine spezielle Brille benötigt, die dafür sorgt, dass die Augen nur jeweils das korrekte Bild sehen. Eine neuere Methode, genannt Autostereoskopie, versucht durch eine besondere Bildschirmtechnik diese Brille überflüssig zu machen. Alternativ kann man den Bildschirm direkt in die Brille integrieren. Ein sogenanntes Head-Mounted Display hat den Vorteil, dass es bei Bedarf den kompletten Sichtbereich des Benutzers abdecken kann ohne dabei selbst allzu groß zu sein. All diese Verfahren haben allerdings einen entscheidenden Nachteil: sie schränken die Bewegungsfreiheit entschieden ein. Wenn der Benutzer sich bewegen will, muß der Inhalt des Monitors angepasst werden, ansonsten leidet der dreidimensionale Eindruck darunter. Abhängig von der Funktionsweise und den Abmessungen des Bildschirms sind üblicherweise nur kleinere Bewegungen durchführbar.
Dies macht es nahezu unmöglich, ein virtuelles Objekt aus einem anderen Blickwinkel, beispielsweise von der Rückseite, zu betrachten. Eine alternative Art der Bewegung muss implementiert werden. Eine bei Computerspielen weitverbreitete Möglichkeit setzt auf eine Kombination aus Maus und Tastatur. Diese beiden Eingabegeräte sind bei fast allen Computern verfügbar. Eine Taste wird gedrückt und die Kamera bewegt sich in eine Richtung, verschiebt man die Maus, verändert sich die Blickrichtung. Präzise Bewegungen können sehr einfach durchgeführt werden, allerdings ist diese Art der Bewegung sehr unterschiedlich im Vergleich zur realen Welt - abgesehen von den Händen muss kein Körperteil bewegt werden. Aus diesem Grund kann es schwer sein in eine virtuelle Szene einzutauchen (Englisch: *immerse*). Das Institut für Computer Graphik und Wissensvisualisierung an der Technischen Universität Graz hat die Möglichkeit, eine glaubwürdige 3D Szene darzustellen, in die man auch eintauchen kann. Durch den Einsatz von mehreren Projektoren und einem optischen System zur Positionsbestimmung kann ein Benutzer sich in einem gewissen Bereich frei bewegen. So muss man nur ein paar Schritte gehen, um ein Objekt von einer anderen Richtung aus zu betrachten. Diese Fähigkeit kann für eine Vielzahl von Anwendungen genutzt werden. Allerdings gibt es auch Beschränkungen die beachtet werden müssen, so kann zum Beispiel die Position nur mit einer mäßigen Genauigkeit bestimmt werden. Zusätzlich sind alle Objekte nur virtuell und können nicht berührt werden.
Diese Diplomarbeit versucht, auf all diese Eigenschaften einzugehen und die Stärken auszunützen ohne die Nachteile und Probleme zu vergessen. Es wird sowohl das Design als auch die Implementierung eines Programmes zur Erstellung von dreidimensionalen Objekten beschrieben. Das Programm trägt daher den Titel *Immersive Mesh Creator* und soll einem Benutzer durch eine möglichst intuitive Steuerung ermöglichen, Skulpturen zu erstellen.

Acknowledgement

This master's thesis was conducted at the Institute of Computer Graphics and Knowledge Visualization at the Graz University of Technology. I would like to thank all those who contributed to this thesis, in particular my supervisors Dipl.-Inform. Dr.-Ing. Sven Havemann, Dipl.-Inform. Volker Settgast and Dipl.-Ing. Marcel Lancelle. Their ongoing support proved to be invaluable.
I would also like to thank my parents for making my studies possible and for assistance with proof-reading.

Contents

Chapter 1

Introduction

1.1 Motivation

3D meshes can be created by using many different applications. Most of these applications such as for example the free *Blender* tool or Autodesk's 3ds Max and Maya are very powerful and enable advanced users to perform almost any task. Usually it is very difficult to learn how to use such an application. There are far too many buttons to be pressed and too many options to choose from. Even after using an application for many years, one can still learn new features constantly. Using only a small subset of the available functionality still requires the user to read the extensive manuals and study tutorials. Hence, there certainly is a need to simplify applications and to allow casual users to use them. Google SketchUp is a very popular application that is not as precise and powerful as the previously mentioned applications. As the name suggests, its emphasis is on sketching rather than professional modelling. Objects are created simply by drawing a shape and pulling it apart. Modifications can be performed in a similar way. Advanced tools require more steps in order to create the same objects. However, the resulting quality could be better. In addition, the user has more ways of modifying it.
Every modelling application faces the same problems: the user has to learn how to navigate in 3D space. The view can either be rotated or moved in a direction. Typically, a combination of both is required in order to reach the target. Moving back and forth between different views can become time-consuming. Hence predefined viewports can be accessed in order to simplify the navigation. Common views are top, front and side. By providing these separate views, the user does not have to continuously change the viewport manually. Instead, the appropriate view can be picked. Very often the screen is divided into separate areas. Each area shows the object using a different viewport. Despite being an efficient way of interacting with a 3D model on a computer, it is not necessarily the best way.
If, for example, some part of a model has to be painted in a different colour, would it not be convenient to just pick up a brush and apply the colour? At least this is what one would do in the *real* world. This idea of going to the part of interest and modifying it, is impossible to implement on a stationary computer. It can only be done in an environment that enables the user to walk around. This way of interacting with an object however leads to certain restrictions. Having to walk in order to accomplish a task instead of just pressing a few buttons on the mouse can be very tiresome. After spending many hours inside of the virtual environment, the users would be completely exhausted. Hence conventional modelling application will not be replaced in the foreseeable future. Nevertheless the technology can most definitely be used for less advanced modelling purposes, like for example drawing or sculpting. With the focus on creating and designing interesting sculptures and drawings rather than highly detailed machines, the modelling experience can be improved significantly.

1.2 The DAVE

1.2.1 Overview

Visualising a virtual environment is commonly done by either a head mounted display or a multi-projector system, with the DAVE being the latter. *DAVE* stands for ***D**efinitely **A**ffordable **V**irtual **E**nvironment*. The first DAVE was built in Braunschweig, Germany in 2003 [FHH03]. Only two years later a new DAVE was built in Graz. It utilises standard hardware components. This makes it affordable when comparing it to commerical systems. The hardware can easily be upgraded, enabling the DAVE to display state of the art graphics. The environment consists of three walls and a floor. Hence four different projectors are required. In order to avoid occlusions caused by the user standing inside the DAVE, a back projection system is used for the walls. Mirrors help to reduce the required space.
The acronym DAVE is supposed to indicate the similarity to the *CAVE* (Cave Automatic Virtual Environment) system presented at the 1992 SIGGRAPH [CNSD93].

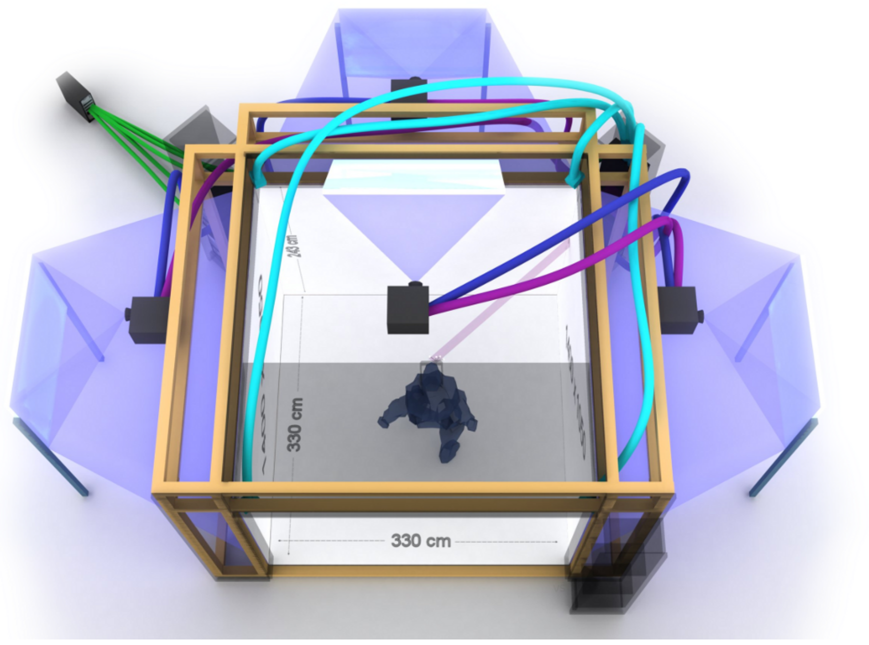


Figure 1.1: 3D model by Volker Settgast and Marcel Lancelle [SL08] depicting the DAVE. The area of the floor is rougly 3,3m x 3,3m, hence the user can move around to some degree. Each projector has a resolution of 1400 x 1050 pixels which enables the DAVE to display high resolution images.

By attaching reflective *markers* to an object, its position inside the DAVE can be determined. Optical cameras use infrared light to locate the markers even if the environment itself is completely dark. Cameras are mounted on the ceiling, one in each corner. Hence it is very likely that at least two cameras can locate the markers. Multiple cameras are required in order to calculate the position in 3D space. Markers found in one image have to be associated with markers found in a different image. The more cameras detect the same markers, the more precise the result gets. Each update requires thorough examination of four different images. Necessary operations are rather time-consuming. Performing a number of updates per second keeps a computer very busy. Hence this optical tracking system developed by Marcel Lancelle at the CGV institute runs on a separate computer. Applications can access the processed data via a network connection.

1.2.2 Immersion

Virtual reality is called immersive if users feel just as *immersed* as they would in the *real world*. There have been studies to determine if an *immersive virtual environment* can improve a user's performance. Randy Pausch et al [PPW97] tried to quantify the effects by letting users perform different actions on a normal computer and in a virtual environment. The results were not particularly different. Proponents however point out that virtual reality enhances the sense of *'being there'*. In the DAVE this effect is achieved by two different techniques: *stereoscopy* and *motion parallax*. Stereoscopy is implemented by using *shutter glasses*. This requires the scene to be rendered twice. Once for the left eye and another time for the right eye. The two different images are projected on a rotating basis. Due to the fact that each image is only visible for a split second, everything appears to be double. The shutter glasses let each eye only see the appropriate image by hiding the rest. Therefore, projectors and shutter glasses are synchronised. The resulting *illusion of depth* makes the whole environment appear more realistic. This increased visual quality justifies the additional computations.

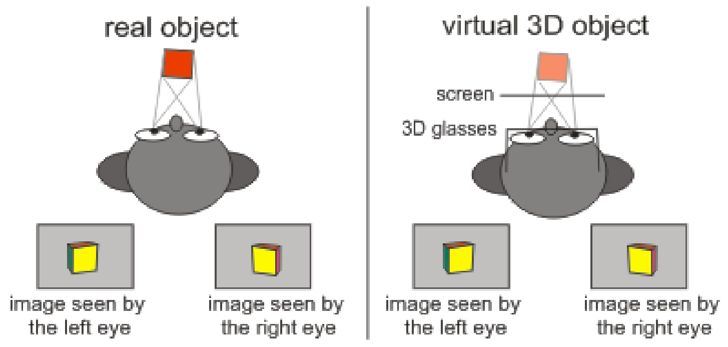

real object
virtual 3D object
screen
3D glasses
image seen by the left eye
image seen by the right eye
image seen by the left eye
image seen by the right eye

Figure 1.2: Diagram by Volker Settgast and Marcel Lancelle [SL08] showing stereoscopic vision. The differences between the two images depend on the distance to the object. The nearer the object is to the viewer, the greater the visible difference.

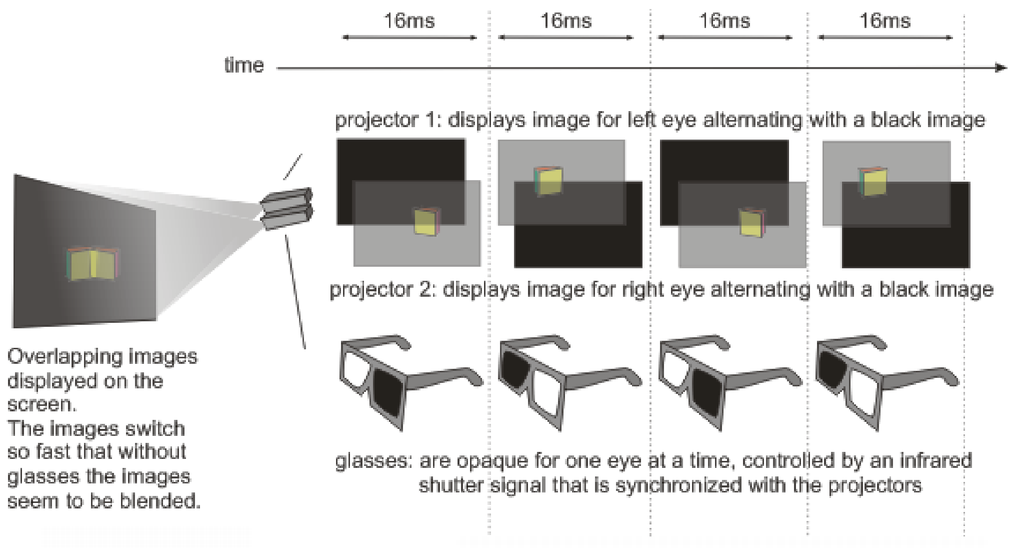

16ms
16ms
16ms
16ms
time
projector 1: displays image for left eye alternating with a black image
projector 2: displays image for right eye alternating with a black image
Overlapping images displayed on the screen.
The images switch so fast that without glasses the images seem to be blended.
glasses: are opaque for one eye at a time, controlled by an infrared shutter signal that is synchronized with the projectors

Figure 1.3: Projector and shutter glasses synchronisation diagram by Volker Settgast and Marcel Lancelle [SL08]. The two images are displayed by a single projector as there is only one projector per wall.

The DAVE is designed to provide this immersive experience for only one user at a time. The scene is rendered depending on the position of the user. The position is retrieved from the optical tracking system. If the user starts to move, the scene will have to move accordingly. As the distances to the walls change as well, the projection will have to be adapted in order to compensate. From the user's point of view everything will appear to be normal, whereas for somebody who is looking at the scene from a different viewpoint, for instance from outside, everything will appear to be distorted. In addition to the distortions, the stereoscopic effect is only valid for the main user as the offset is calculated based on the viewing direction.
Stereoscopy leads to the second technique that improves the immersion. Illusion of depth has a greater influence on objects that are near to the viewer. Hence by moving the head to the left, near objects will move more than objects that are far away. This effect has already been utilised in countless applications such as old two dimensional computer games. Just by comparing the speed, one can immediately distinguish near and far objects.
To complete the immersive experience, a surround sound system is available. With the ability to play sounds from almost any direction, it will feel like a real environment. The sounds do not even have to be particularly loud and distracting. A subtle clicking sound if the user is performing an action can already be enough feedback. Very often the audio part of an application is neglected, even though it does not require too much effort to implement. It is sometimes considered an optional feature and therefore gets postponed as other parts seem more important. Applications running in the DAVE are recommended to utilise the audio system, as the addition of sounds can have a particularly positive influence.

1.3 Problems and Challenges

1.3.1 Limited Input Capabilities

The user is standing inside a virtual environment and does not have access to the usual input devices like mouse and keyboard. There are certain criteria that input devices in such an environment have to fulfil. They should neither restrict the movement of the user nor be too complicated to operate. As it can be rather dark, the user has to be able to use the devices without looking at them. Preferably, the devices should be designed to be operated with a single hand. These requirements lead to a significantly reduced amount of buttons that can be placed on such a device and still allow the user perform complicated tasks.
One way of increasing the capabilities of the device is to utilise an optical tracking system. If markers are attached to a device, additional input parameters can be used such as position and rotation. Tilting the device can trigger off an action, and moving it back and forth a different action. Movements like these can be done much more intuitively than pressing certain combinations of buttons, and will greatly increase the usability if implemented properly.
The controllers used by Nintendo's Wii console have a special motion sensing capability which is used to record the movements of the player. Accelerometers can detect motions conveniently and independently of the position of the user. While the optical tracking system of DAVE has some advantages when recording the absolute orientation of an object, motion detection is rather limited. Very fast motions cannot be detected due to hardware limitations. By adding extra motion sensors to an input device in the DAVE, one could have the best of both worlds.

1.3.2 Precision

The *optical tracking* used by the DAVE is based on infrared cameras. As such, the CCD sensor's resolution and quality greatly influences the precision. The closer a marker is to the camera, the more precise the result will get. The amount of pixels per marker depends on the distance. A near marker will occupy a significant amount of space in the camera's image. Moving the marker will result in a correspondingly large change in the image and even small movements will be visible. By increasing the distance to the camera, the number of pixels will decrease accordingly up to the point where just a couple of pixels are left. Each pixel consists of three sub-pixels. These sub-pixels represent a colour. However, the colour of a pixel is not important as the image will appear black and white. Hence the sub-pixels can be used independently. One sub-pixel equals the smallest movement that can be registered. If the marker is only two pixels wide in the image, moving it by one pixel is already half of the size of the marker.
With a camera mounted in each corner of the room, there should always be at least one camera that provides reasonable values. A convenient side effect of such a set-up is that markers occluded by something will most likely be still visible by one of the cameras. However, in order to get reliable data, at least two cameras have to spot the markers. Each pixel represents a ray originating from the camera. If a marker occupies one of these pixels, it could theoretically be anywhere on this ray. Hence these rays have to be intersected with rays originating from different cameras. If more than two cameras detect the same markers, the optical tracking system can decide which images are used in order to maximise the precision.
One more aspect is important when it comes to optical tracking. 3D applications usually render sixty images per second in order to get smooth animations. Not all cameras have the capability to capture sixty images per second. The cameras of the DAVE are connected via FireWire and can transmit up to sixty four images per second. LEDs are installed next to the cameras and illuminate the DAVE with infrared light. Markers reflect this infrared light and enable the cameras to detect them. However, the LEDs are not particularly bright. The exposure time of roughly four milliseconds results in a dark and blurry area in the image if a marker is moved too fast. Detecting the position of this marker is

going to be difficult.
Processing all of the images is very time-consuming. The tracking system currently operates at a rate of fifty updates per second.

1.3.3 Estimating Distance to Objects

The whole scene is projected onto walls. This means that no matter where the user is standing or how he is holding the input devices, everything seems to be behind the user's arms. When trying to reach behind an object, there is no appropriate visual feedback. Even if there actually is an object between the eyes and an arm, it will always be occluded by the arm. This leads to a major drawback when using such a method to visualise the environment: it might be very difficult to estimate the distance to an arbitrary point in the scene.
As soon as the viewpoint of the user moves from one side to another the parallax starts to take effect. Objects in the distance appear to be moving more slowly than objects that are close to the user. Usually the user is not standing completely still, which makes this effect somewhat ubiquitous. Stereoscopy appears to have more influence on objects that are near the user. Nevertheless, it can sometimes be difficult to determine the correct distance and size of an object. Certain properties of a surface like a texture can help to visualise its distance. By comparing the size of the texture in two different points it can be concluded which of them is the nearer one. This requires spreading the texture evenly over the surface, and that is something that has to be kept in mind when creating an object, otherwise it will confuse the user. If one object is twice the size of another object, the texture should be set accordingly.
A surface texture also helps the eyes to merge the two separate images required by stereoscopy. If an object is covered by a uniform colour, it is difficult to find corresponding points in each of the pictures.

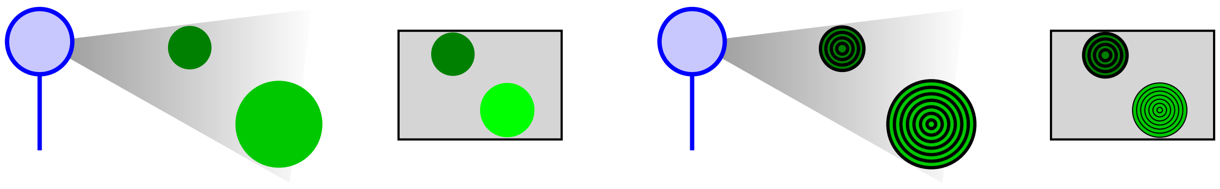


Figure 1.4: As long as objects have no visible surface structure, it is difficult to tell how far away they actually are. This series of figures tries to demonstrate the problem. The first figure depicts the actual scene whereas the second figure shows the same scene from the viewpoint of the user. The two spheres of different sizes appear to be almost identical due to the perspective. By adding a uniform texture, in this case equidistant circles, one sphere will appear to be smaller and therefore nearer because of the coarser texture.

1.3.4 Missing Haptic Feedback

Haptic feedback refers to a technology that enables the user to feel objects by applying force or vibrations to the skin. Usually motors will use mechanical motion to simulate the touch. Well known devices are gloves with integrated haptic feedback or special pens. They provide different kinds of sensations and can be used for many applications. The glove needs either long cables or heavy batteries. A haptic pen requires a specially constructed mounting and might be difficult to move because of its size. Both of them limit the movement of the user to some degree. However, a user might be willing to accept this restriction in order to get the ability to touch objects.
Unfortunately the DAVE in Graz does not have any haptic devices. The user is not able to touch any objects and will always be reminded that the objects he is creating are not really there. Faster

computers will only increase the visual quality in the future, but the lack of haptic feedback will still remain.

1.4 Possibilities

1.4.1 Drawing

Inside a virtual environment it seems natural to draw a line by just moving an input device from one place to another. Multiple lines or other primitives can be combined to form interesting objects. In a two-dimensional drawing the perspective cannot be changed whereas here the user can walk around and take a look at his creation from different angles. This greatly increases possibilities. Being able to look at a drawing from behind can make it seem much more immersive. But that will require the artist to create a drawing that looks good from any perspective.

CavePainting: A Fully Immersive 3D Artistic Medium and Interactive Experience

Daniel Keefe and his colleagues [KFM*01] presented a 3D painting application running in a four wall immersive virtual reality system called a Cave Automatic Virtual Environment (*CAVE*), similar to the DAVE. Scenes are created by arranging virtual 3D strokes in space. Several types of brushes are available, enabling the user to draw many different things, while trying to make it natural for artists and novices to use. The user can draw a few strokes, walk back and take a look at them, making use of the CAVE's capabilities.

Drawing on Air

While it can be used intuitively, there definitely is a lack of control. This paper by Daniel Keefe et al [KMZ*07] describes a way of drawing in a 3D space by hand without losing precision. But instead of being in a CAVE environment the user is looking at a stereoscopic desktop display. A haptic device is used for a one-handed drawing and an additional tracked marker for a special two-handed tape drawing method. This results in an easy to learn, yet powerful method of drawing in 3D space.

DAB: Interactive Haptic Painting with 3D Virtual Brushes

Bill Baxter et al [BSLM05] presented an interesting way of using a haptic painting device setup for creating 2D drawings at the 2005 Eurographics conference. Using the haptic abilities of the pen, the programme can simulate 3D pens by emulating the sensation of applying brush strokes to a canvas. The input device will provide force feedback. Different pen and canvas properties can be set, making it a powerful tool.

Interactive Mediated Reality

Enhancing the drawing ideas leads to the next paper [GBGS05]. Instead of just drawing on air or onto a single planar canvas, real and virtual elements can be combined. The user can paint on real objects and add virtual objects to the scene. Because the paint itself is only virtual, the same objects can be used multiple times.
Stable tracking is a definite must, as the real and virtual elements have to move synchronously when the user is changing the viewing point. Being an augmented reality application, it is not very well suited for the DAVE. The setup described in the paper uses a head mounted display, whereas the DAVE uses projectors making it difficult to colourise real objects.
Deepak Bandyopadhyay et al [BRF01] presented a technique called Shader Lamps. Real objects covered by white colour can be placed on a surface. Cameras detect the orientation of the objects. Multiple projectors illuminate the objects from different angles. Hence the plain white colour can be replaced by an arbitrary painting.

1.4.2 Modelling

Instead of creating a drawing by laying several independent strokes on top of each other it is also possible to produce connected models. For instance, by removing certain parts, a cube can slowly be turned into a sculpture. It immediately becomes apparent that operations like that require a totally different set of tools.
To create 3D models on a computer, input devices like a mouse and a keyboard are required to perform complex operations in special modelling programmes. In a virtual environment like the DAVE the user will not have access to any of these tools and due to the limited precision cannot achieve the same performance as above.

The over-sketching technique for free-hand shape modelling in Virtual Reality

With precision being an issue, the outcome of an operation might not always be the expected result. Fabio Bruno et al presented a way of modifying curves and surfaces after they were created [BLRC03]. Their over-sketching technique for free form modelling makes it possible to refine parts of a curve by simply substituting them with new ones. Previous versions of the curve are visible to some degree, allowing the user to see the differences that were caused by the changes. As soon as the modifications are complete, the old curves can be removed.

Sketch-Based Procedural Surface Modeling and Compositing Using Surface Trees

Ryan Schmidt and Karan Singh [SS08] presented a paper at the 2008 Eurographics conference combining the power of procedural with the simplicity of sketch based modelling. Multiple edit steps are combined in a hierarchy called a surface tree, similar to a layer technique that is used in 2D image manipulation programmes. Linked copies can be created, enabling the user to re-use certain parts. Changes to one instance will have an effect on all linked copies. Thanks to the procedural modelling approach, all parameters can be modified at any time.

SKETCH: An Interface for Sketching 3D Scenes

With SKETCH [ZHH06] a user is supposed to be able to easily create 3D models using nothing but a mouse and gestures. The idea of sketching things with pencil and paper gets transformed into 3D space. Re-drawing something in order to look at it from a different angle is no longer necessary. Easy to learn rules enable the user to place geometry in the scene. The geometry can either be added to or subtracted from the existing scene using the constructive solid geometry technique.
A special rendering style tries to emphasise the sketchy character of the models.

Teddy: A Sketching Interface for 3D Freeform Design

The Teddy programme created by Takeo Igarashi and his colleagues [IMT06] provides a unique way of creating 3D models by drawing freeform strokes on the 2D screen. Depending on the silhouette areas get either fat or thin. It is especially well suited for creating stuffed animals or any other round object. A gesture recognition can interpret user commands making the whole interface very easy to learn. Certain parts of an object can, for instance, be removed by cutting it in half with a single stroke.
The application was written in Java and can convert drawings into polygonal meshes in realtime.

iWIRES: an analyze-and-edit approach to shape manipulation

The iWIRES technique by Ran Gal et al [GSMCO09] uses a special two-phased approach. First the mesh is analysed and the so called wires are placed on sharp edges. In order to fully describe the structure of a mesh, the wires need to be interconnected and form groups. If, for example, two wires

lie on the same plane, they will continue to do that even if one of them gets deformed. Changing one of the wires will influence others. These wire relations have to be chosen wisely. Too many of them might add unwanted dependencies, whereas too few can render the whole technique inoperable. As the relations are persistent, this step only has to be executed once.
The second step consists of editing operations that will deform the mesh by manipulating the wires. Changes are propagated to all the other wires, which completes the transformation. What makes the technique special is the fact that these operations will not destroy the original mesh's characteristics. A circle will stay a circle, even if the mesh itself gets stretched.
In order to work, a mesh with easily identifiable edges has to be provided. Usually man-made objects like tables or chairs are well suited. The wire detection phase will fail on anything that lacks sharp edges, like an animal for example.
By introducing a sketching operation, selected parts of a mesh can be rearranged in a very intuitive way. It might work very well in a DAVE-like environment, as the user no longer has to worry about preserving the structure of the mesh.

1.4.3 User Interaction

There are many different ways of dealing with user input in a virtual environment. Besides pressing buttons on an input device, usually its movement can either be tracked by cameras or other sensors enabling the use of many degreees of freedom (position, rotation, acceleration, ...). Utilising its full potential can be a tricky task. Depending on the application, certain approaches might be better than others. Sometimes a gesture based system will work well, whereas another application will only require the absolute position of an object.
As soon as an application gets slightly more complex and provides many different tools, some sort of user interface will be necessary. Two dimensional interfaces usually rely on windows filled with buttons and other widgets. Adapting such an interface to a three dimensional environment is important and has a huge influence on the usability. If an application is too complicated to use, not many people, except its developers, will be able to work with it. Having to deal with too many tiny widgets with lots of text can be just as confusing as pressing many different combinations of buttons in a virtual environment. That is why a considerable amount of work should be put into creating an easy to use interface.

Geometric Modeling Using Six Degrees of Freedom Input Devices

When using traditional input devices like mouse and keyboard, 3D tasks have to be decomposed into a series of 1D and 2D tasks. Jiandong Liang et al [LLG93] implemented a 3D modelling system called JDCAD. A tracking system is used to monitor the head's as well as a bat's position. The user therefore has access to a six degrees of freedom input device. This input device allows the user to directly manipulate objects in 3D space. In addition to this input device, the second hand can be used to operate a keyboard. Hence the user has access to the capabilities of a keyboard as well as the flexibility of a six degrees of freedom input device. A standard monitor is used instead of a head mounted display. By avoiding the physical burden of wearing a head mounted display, the system can be used for a significant amount of time. Special interaction techniques such as selection menus were implemented. The modelling system proved to be a very efficient way of interacting with the 3D scene. Mechanical components could be created in roughly one-fifth to one-tenth of the time that would be necessary when using commerically available systems.

Working in a Virtual World: Interaction Techniques Used in the Chapel Hill Immersive Modeling Program

In the Chapel Hill Immersive Modeling Program (CHIMP) created by Mark Mine [Min96] a user is supposed to be able to create and manipulate models while immersed within a virtual world. Special bats serve as input devices. It is possible to operate CHIMP using either one or two hands. By simply pointing at an object, it can be selected. Depending on the subsequent movement, this object gets manipulated. More advanced operations can be performed by accessing control panels. These control panels resemble standard graphical userinterfaces and contain buttons and other widgets. A special control panel contains miniature represention of all objects that are currently available. Such an object can be added to the scene by dragging it to the appropriate position.

SPIDAR G+G: A Two-Handed Haptic Interface for Bimanual VR Interaction

Jun Murayama et al proposed a new haptic interface [MBA*04]. The user should be able to use both hands to intuitively rotate and translate virtual objects and get force and torque feedback. Grabbing specially mounted objects, a programme can take advantage of six degrees of freedom per hand. Studies have shown that it can be easier to perform complex tasks with two hands.
Unfortunately the user has to grab the input devices all the time, disabling the possibility to move around in a virtual environment and thus making it unsuitable for the DAVE.

Real Time Hand Tracking and 3D Gesture Recognition for Interactive Interfaces using HMM

A system that might work better in the DAVE was presented by C. Keskin et al [KEA03]. The user has to wear coloured gloves that are tracked by multiple cameras. Coordinates are obtained by processing images using a shape from stereo method [FDSB04]. A hidden Markov model is used to recognise gestures quite accurately. But in order to work, the vision part has to be reliable. To use such a setup in the DAVE, a considerable amount of effort will have to be put into perfecting it.

VisionWand: Interaction Techniques for Large Displays using a Passive Wand Tracked in 3D

Xiang Cao and Ravin Balakrishnan present an alternative mechanism for interacting with large displays [CB04]. A wand is tracked by a computer vision based system, being the only means of providing input. There are no buttons available. All actions have to be triggered by moving the wand. Combined with a special graphical user interface like pie menus, a wide range of tasks can be accomplished. This buttonless way of interacting with a large scale display proved to be easily understandable.

Nintendo Wii Remote

In 2005 Nintendo announced their new Wii gaming console. It uses a unique controller called the Wii remote, or *Wiimote*, to interact with the user. This controller has the ability to detect motion using accelerometers. The built-in infrared camera enables the user to point at any position on the screen. Recently, the controller can be upgraded with the so called Motion Plus module. This module enables the controller to detect its orientation. In case one-handed input is not sufficient, Nintendo provides a second input device called the Nunchuk. It has its own accelerometers and can be connected to the main controller. These capabilities can be used for many different applications. As the controllers use a standard Bluetooth connection to communicate with the gaming console, connecting the controller to a normal PC is rather easy. The comparatively cheap price makes it especially interesting.
Thomas Schlömer et al [SPHB08] tried to use the Wii remote to detect motions and recognise gestures. The motion, represented by a vector, needed to be analysed. They chose to use a hidden Markov model, preceded by a filtering process. This enables the user to define own gestures, and resulted in a very reliable gesture recognition. After a training phase was complete, up to 95 percent of all gestures were successfully recognised.
A different application utilising the Wii remote was devised by Torben Schou and Henry Gardner [SG07]. Instead of using the accelerometers, they accessed the infrared camera. This camera requires

a sensor bar in proximity. The bar merely consists of two LEDs that are separated by a certain distance. By measuring this distance, the controller can roughly estimate how far away it is from the sensor bar. As the controller's camera however, has a limited field of view and resolution, moving too near or too far away causes problems. Therefore, in a large environment, multiple sensor bars might be necessary. Torben Schou and Henry Gardner wrote a special driver that is capable of processing multiple sensor bars. Hence the wii remote can efficiently be used in their virtual reality theatre.

Twister: a space-warp operator for the two-handed editing of 3D shapes

Using Twister [LKG*03], the shape of an arbitrary object can be modified in a unique way. The position as well as the rotation of each hand is processed. As soon as two separate points on the surface of an object are selected, the user can start to translate and rotate these points by moving the hands, making the object bend and twist. These transformations can alter the shape of an object quite significantly. The smoothness of the surface is preserved by adaptive surface subdivision.

1.4.4 Common Mesh Representations

While there are many possible applications that can be created for the DAVE, they all have one thing in common: the drawings or models created by the user have to be stored and processed in some way. These 3D representations usually consist of some basic elements like polygons or points. Today's graphics processing units are highly optimised for drawing a large amount of triangles. As a consequence any triangle based data structure would seem an obvious choice. There are however, many different kinds of representations for different purposes.

Polygon Mesh

A *polygon mesh* is the most common way of representing geometry in 3D computer graphics. Given lists of vertices, edges and faces, any shape can be constructed. Vertices correspond to positions in 3D space and can have additional parameters like colours or texture coordinates. Edges are formed by connecting two vertices. Faces consist of closed lists of edges. Usually faces will either be triangles or quads.
The rendering process is relatively straightforward. The graphics processing unit just needs access to these data structures. Although when using *OpenGL*, the Open Graphics Library, a few things have to be kept in mind. To reduce the amount of data, the graphics processing unit can automatically ignore polygons that are not facing the viewer. This technique is called *backface culling* and can help to speed up the rendering process. As a consequence it is important in which order the edges are traversed. The normal vector of a face is determined by two adjacent edges. If these edges are traversed in a different order, the normal vector will point in the opposite direction. Hence three vertices can form two separate triangles. These triangles are in the same position, but face in opposite directions. However, it is not always necessary to distinguish between these two triangles. Depending on the object, the backface culling mode might be unnecessary.
To make it easier to get from one face to another, neighbour information can be added. While it is not important for the rendering process, it is essential when trying to modify the mesh.

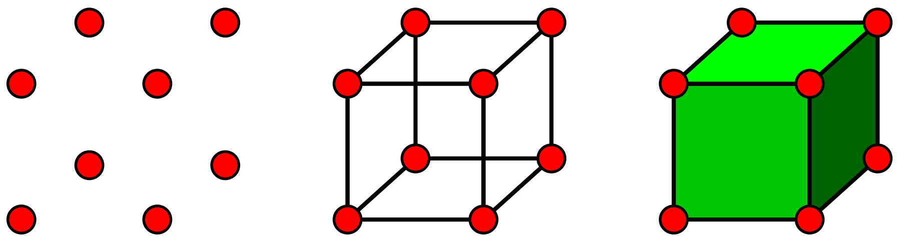


Figure 1.5: Simple example showing only vertices in the first picture, connecting them in the second picture by adding edges, and finally combining edges to faces.

Point-Sampled Geometry

Point based rendering is a completely different way of representing geometry. A cloud of points is placed on the surface of an object. If this cloud is dense enough, no holes will be visible and the points form a closed surface. The amount of detail is limited by the point size. A sufficiently small size will enable a very detailed surface structure. The major advantage of this kind of data structure is its simplicity. No extra information like the connectivity has to be stored. Hence in order to get nice looking objects the number of points has to be quite high and is independent of the surface's complexity. A single plane for instance can be represented by just two triangles, whereas point based geometry needs a whole cloud of points. Points do not necessarily have to be circles but can be of any elliptic shape.
Mario Botsch and his colleagues presented a way of efficiently rendering point sampled geometry

[BWK02] using a software implementation. A hierarchical octree based representation reduced the memory requirements and proved to be fast enough for real-time applications rendering several millions of points per second. To further increase the performance, the GPU's capabilities have to be utilised. As the GPUs are not optimised for this kind of rendering technique, some problems have to be dealt with. A compromise between performance and visual quality has to be made. Mario Botsch et al tried to minimise the trade-off between quality and efficiency [BHZK05]. By using deferred shading and floating point rendering targets, shading artifacts can be avoided. The resulting output almost matches the quality that can be achieved with a software rendering process, yet being considerably faster.
Pointshop 3D by Zwicker et al [ZPKG02] is a powerful system for interacting with point based geometry. It can be used to map textures onto surfaces by specifying feature points on both the geometry and the texture. The geometry itself can be modified by using normal displacement.

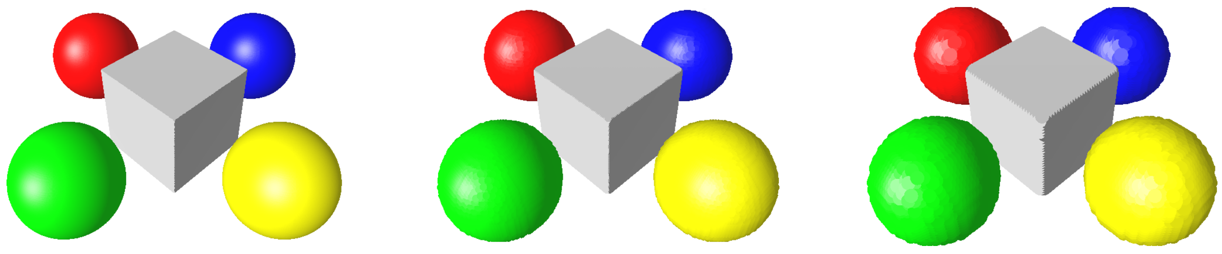


Figure 1.6: Example of point based rendering using a simple implementation. The point sizes are 1, 5, and 10, making the first image consist of approximately one hundred times as many points as the last image. The bigger the pointsize is the more artifacts are visible.

Generative Modelling

Generative modelling is an alternative way of describing a shape. Instead of storing geometric primitives like triangles in a polygon mesh, lists of operations are used to build complex objects. By sequentially processing these functions, any arbitrary object can be created. Sven Havemann [HF05] devised a programming language that is specifically designed for this purpose. It is called the *Generative Modeling Language* (GML) and uses a stack-based postfix notation. By utilising techniques such as subdivision surfaces, shapes can be described in a very basic manner and nevertheless contain smooth surfaces. Many different operations are provided to allow fast and easy mesh creation. Parts of an object can be reused multiple times. By changing the original part, all the other parts are updated accordingly. Therefore this way of describing a mesh is very powerful when it comes to objects that contain multiple instances of the same parts. The so called *Procedural Cathedral*, a model depicting the Cologne cathedral, consists of slightly more than one hundred kilo bytes of GML code. Nevertheless this code generates several million triangles. If the same geometry were to be stored in a normal polygon mesh, the file size would be far greater. The mere amount of triangles would make any kind of modification very difficult.

Implicit Surfaces

Almost all mesh representations rely on basic primitives such as lines, polygons or spheres for example. It is often very difficult to implement smooth surfaces or enable objects to be deformable using these representations. *Implicit surfaces* however try to address these problems by creating isosurfaces. Isosurfaces are surfaces that represent points of the same value. They can be generated by defining different field functions. One popular technique is called *Metaballs.* Control points can be placed in 3D space. These control points have a radius parameter that influences the size of the spheres (or balls) that are created around them. By combining different primitives, a control point does not always have

```
0.7 !w
1.4 !d
1.0 !h
0.05 !t
0.06 !ws

:h 0.25 mul !h_offset
:h :h_offset sub !hs

A24−Fraser.Tools begin
0 :h_offset :d 2 div vector3 :t :hs :w front
0 :h_offset :d −2 div vector3 :t :hs :w front

0 :w 2 div 0 :d 2 div vector3 :ws :h leg
0 :w 2 div 0 :d −2 div vector3 :ws :h leg
1 :w −2 div 0 :d 2 div vector3 :ws :h leg
1 :w −2 div 0 :d −2 div vector3 :ws :h leg

:w −2 div :ws sub :h_offset 0 vector3 :t 0.5 mul :d :hs side
:w 2 div :ws add :h_offset 0 vector3 :t 0.5 mul :d :hs side

0 :h_offset 2 mul 0 vector3 :t 4 div :d :w frame
end
```

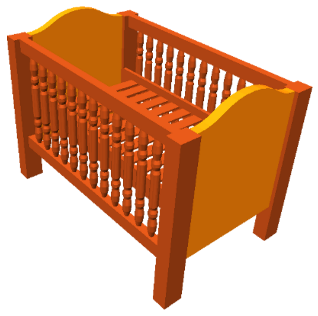


Figure 1.7: A small GML example showing IKEA's Diktad bed. The different parts (front, leg, side, frame) are created as separate objects and put together. Values can be stored in variables using an exclamation mark and the name. Accessing them is done by using a colon. The bed's dimensions can be altered by changing the values of the corresponding variables. Based on these variables, the parts are placed in the scene.

to be a sphere. A threshold parameter defines at which relative distance primitives begin to merge. Control points can either be set to addition or subtraction. Hence pieces of an object can be cut out by placing a subtracting point nearby.

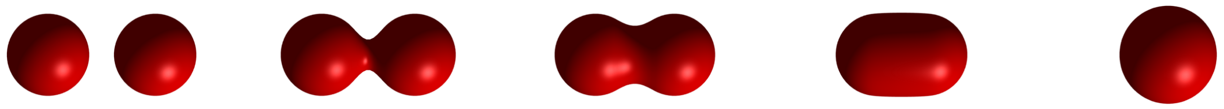


Figure 1.8: The Blender 3D graphics application has built-in metaball support. As soon as the two spheres get nearer than a certain threshold, they start to form one connected object.

Convolution surfaces are a more advanced way of representing meshes. A smooth surface is placed on a shape that can be described by using various primitives likes lines or polygons. No matter how the primitives move, the surface will remain smooth. Andrei Sherstyuk [She99] showed a way how this technique can be utilised in a shape designing application. It is sufficient to describe an object by defining its skeleton. This skeleton will eventually turn into the desired object if the radius parameter is set accordingly. The skeleton represention is also very well suited for animations, as described by Jules Bloomenthal et al [BS91].

Chapter 2

Design of the Modelling Application

2.1 Description

As shown in previous chapters, the DAVE's capabilities can be used to perform various tasks. Any kind of action that involves sketching and does not depend on precise and complicated movements seems to be well suited. The primary goal of this programme is to provide tools for creating and modifying meshes, utilising the 3D environment provided by the DAVE. As the user is going to be able to look at a mesh from different angles, the surface must not contain holes. Therefore the user can focus on modelling rather than preserving the surface structure by continuously closing holes and gaps. Modelling usually involves many different and sometimes complicated operations. With this application a user should be able to change the shape of an object by intuitively moving an input device. A flat surface for example can get a dent if the user grabs a piece and pulls it into a certain direction. A new and separate object can be created at another position by performing a different kind of action. Therefore the power of sketching and sculpting are combined, resulting in a very flexible tool. It can be used for many different scenarios. Some of them might work better than others, hence a selection of different modelling tools was devised. They can be used to demonstrate the possibilities and enable the user to quickly create meshes.
These modelling tools require a special mesh representation that provides the necessary functionality. Reusing and enhancing an already existing implementation does not have many advantages, besides a probably better performance. They all lack certain features, and adding them is sometimes more complicated than rewriting everything from scratch. Hence an essential part of this application will be to have a reliable mesh representation that supports all the required functionalities.
Besides these aspects concerning the mesh, there are far more things that are of great importance and influence the whole user experience. Being able to walk around and look at one's creation from different angles already provides a great deal of immersion. This experience can be enhanced by adding an environment that does not only change the appearance of the surroundings but also serves as a means of interacting with the application. It will be like an artist's workroom in which all the tools are ready to be used. Therefore it might appear familiar to the *real* world and enable new users to start modelling right away without getting long and complicated instructions. This part should not be neglected as it has a great impact on the usability. If an application is too complicated to use, not many people will try it out. Hence putting work into the human computer interaction part is at least as important as the modelling part.
This chapter describes how all of these requirements are handled. The resulting design of the modelling application contains important aspects such as user interaction and an overview of all available tools. More specific details like for instance the implementation can be found in subsequent chapters.

2.2 Mesh Representation

2.2.1 Half-Edge Data Structure

One way of representing a polygon mesh is to use a *half-edge* based structure. If two vertices are connected, there can be two separate edges pointing in opposite directions, each of them being a half-edge and belonging to a different face. Such half-edges are called mates and make it possible to jump to adjacent faces. Besides storing the mate, a half-edge has access to its predecessor and successor. If, for example, a polygon is a triangle, the successor's successor will be equal to the original edge's predecessor, as the face's edges form a ring. By knowing just one edge the face has full access to all of its edges.

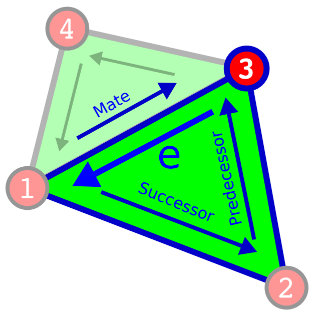


Figure 2.1: Properties stored at edge e: successor, predecessor, mate and the originating vertex.

Usually an edge is defined by the two vertices it is connecting. A half-edge only has to store the vertex it is originating from. The second vertex is determined by the succeeding edge. By changing the successor, an edge will point to a different vertex, making it easy to build faces.
Storing the predecessor might appear to be unnecessary. By following the edges of a face, the predecessor will be reached anyway. Especially when limiting the possible face types to only triangles, it will not make much of a difference. The increase in speed is marginal at best and does not really justify the additional memory consumption. Connecting edges in both directions can act as a consistency check and help to detect errors caused by mesh modifications.

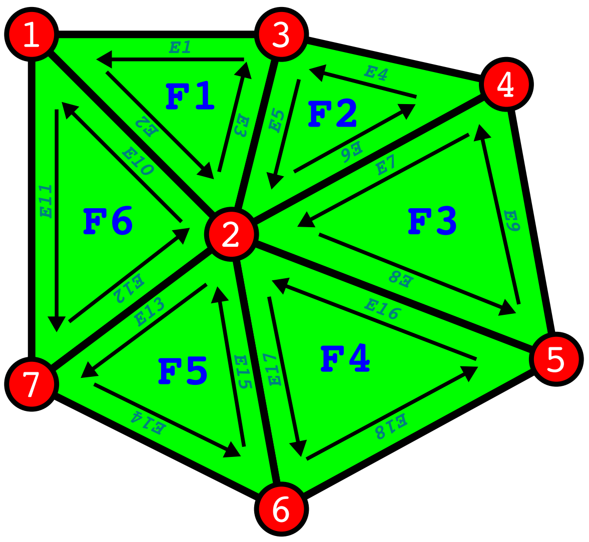


Figure 2.2: Half-edge mesh example

Face	Edge	Vertex	Mate	Successor	Predecessor
F1	E1	V3	-	E2	E3
	E2	V1	E10	E3	E1
	E3	V2	E5	E1	E2
F2	E4	V4	-	E5	E6
	E5	V3	E3	E6	E4
	E6	V2	E7	E4	E5
F3	E7	V4	E6	E8	E9
	E8	V2	E16	E9	E7
	E9	V5	-	E7	E8
F4	E16	V5	E8	E17	E18
	E17	V2	E16	E18	E16
	E18	V6	-	E16	E17
F5	E13	V2	E12	E14	E15
	E14	V7	-	E15	E13
	E15	V6	E17	E13	E14
F6	E10	V2	E2	E11	E12
	E11	V1	-	E12	E10
	E12	V7	E13	E10	E11

Table 2.1: Half-edge mesh example. The faces can pick any of their edges.

2.2.2 Operations

2.2.2.1 Mesh Traversal

Face Iterator

Given an edge it is often necessary to iterate through the edges belonging to a face, hence the name. As all edges store their successors and predecessors, it can easily be executed. This is one of the most basic operations and is used very often to accomplish more complicated tasks. The default way of defining a face is to traverse edges in counter-clockwise (CCW) order, whereas clockwise (CW) order flips the direction.

- **faceCCW**: return next edge
- **faceCW**: return previous edge

Vertex Iterator

Combining the face iterator with the mate information leads to the next operation. A vertex can have multiple outgoing edges, each of them belonging to a different face. By accessing an edge's mate, one can jump to the adjacent face. This edge mate however is pointing away from the vertex of interest. The face iterator can now help to select the right edge.

- **vertexCCW**: faceCW followed by edge mate operation
- **vertexCW**: edge mate followed by faceCCW operation

This operation can cycle through all outgoing edges in either CW or CCW order. As a result it can be used to get a list of all edges originating from a vertex. By adding these edges' mates to the list, all edges connected to a vertex in either direction can be retrieved.
Calling one of these operations multiple times in a row will eventually return the original edge. Therefore the starting edge needs to be compared with the result of each step in order to detect a completed

cycle.
By processing such a list of edges, many useful properties can be extracted:

- **getVertices**: get all adjacent vertices
- **getEdges**: get all connected edges
- **getFaces**: get all adjacent faces

The face and vertex iterators need an edge to work, nevertheless a single vertex is enough to access all connected edges and faces. Each vertex only needs to store one of its outgoing edges to provide this access. When modifying a mesh one has to make sure that this single edge is still valid.

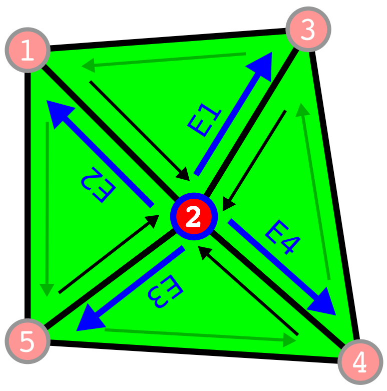


Figure 2.3: Iterating around the vertex 2 will result in the edges E1 to E4.
Getting from E1 to E2 can be achieved by a single vertexCCW call, whereas vertexCW will result in E4.

Border Edges

In some cases border edges might be of particular interest. The mere lack of a mate makes an edge a border edge. A border vertex needs to be connected to at least one border edge. In order to find out if a vertex is lying on the border, one needs to check all edges that are either originating from or leading to this vertex.
The same procedure can be used to iterate along border edges. A border vertex can only have one outgoing border edge. By following this border edge to the next vertex, the next border can easily be found.
Due to these attributes no further information needs to be stored.

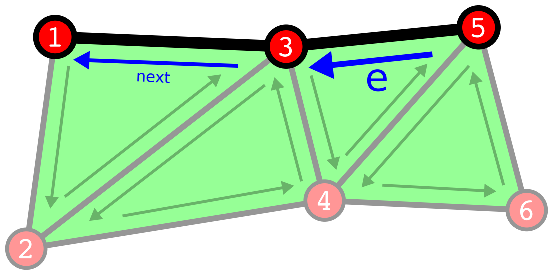


Figure 2.4: Given border edge *e*, the next border edge can be found by checking all edges originating from vertex 3

Border vertices have a special behaviour when using vertex iteration. As these operations depend on the mates of the edges, it becomes apparent that if an edge does not have a mate and the iterator will fail.

2.2.2.2 Face Creation

When creating new faces, the mesh has to stay consistent. Making sure that all adjacent edges are set as mates is an important task for face creation operations. As faces consist of edges and vertices, these have to be created separately:

- **addVertex(P)**: creates a vertex at the specified position
- **addEdge(V)**: creates an edge originating from the given vertex

Vertices and edges created by these two operations are not yet part of the mesh. In fact, as long as edges are not connected to other edges, they do not even have a destination to point at. Giving edges a purpose and connecting them with each other is the job of the face creation operations. There are several operations available, each of them taking different parameters. These parameters can either be vertices or edges. They are used as a means to connect the face to the existing mesh. The operations are:

- **addFace(V, V, V)**: span face between the three vertices
- **addFace(E, E)**: create face using the two edges as sides
- **addFace(E, V)**: use edge as a base and connect it with vertex

All of these operations will create three new edges and a face, if the provided parameters are valid. Edges used as arguments for an addFace operation already belong to other faces and therefore need to be border edges. To make sure that all neighbour information is stored, a special operation is necessary. Given two vertices, an edge connecting them needs to be found. As there might be two edges, the direction is defined by the order. The edge is retrieved by getting all edges originating from the first vertex. Only one of them is going to point to the second vertex.

2.2.2.3 Edge Split

In case an edge is too long, it can be split into two separate edges. An extra vertex is added in between. By connecting this new vertex with all adjacent vertices, the original faces are no longer valid. They must be modified to use the newly formed edge. As their areas are reduced, new faces have to be created in order to fill the empty space.

Only the two directly connected faces are influenced by the *edge split* operation. Their neighbours will not notice any change. It is well suited for refining a mesh's tessellation without changing its original shape. One problem is the fact that some of the newly formed triangles might have a smaller area than others, requiring extra handling.

2.2.2.4 Edge Collapse

If one of the triangle's edges is short compared to the other two, the area will be small. Therefore removing the triangle will not influence the mesh's appearance very much. The resulting hole can be filled by collapsing the selected edge and forcing the two connected vertices to merge, making one of them obsolete.

A maximum number of two faces and the accompanying edges have to be deleted. After merging the vertices, one of them becomes obsolete and can be removed as well. Adjacent faces will be deformed, hence this operation has to be used carefully. It might produce an unwanted side-effect if an important edge gets modified just because a nearby edge collapses. Important edges can be border edges or sharp edges. Changing either one of them will be clearly visible. Hence preliminary checks have to be implemented.

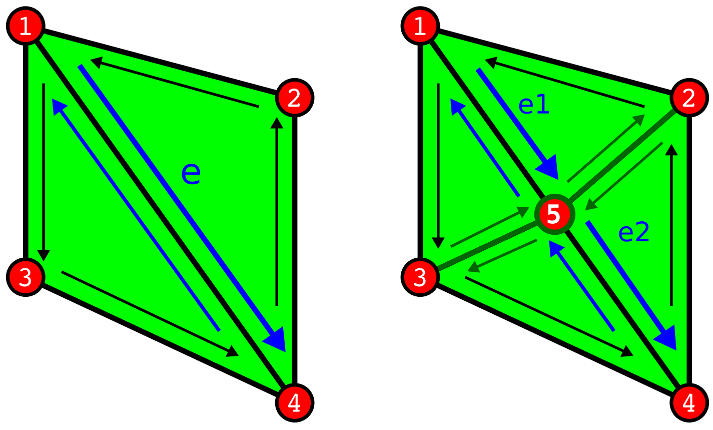


Figure 2.5: Splitting edge e will create a new vertex on its centre (5) and new edges connecting it with all the adjacent vertices. The number of edges and faces has doubled.

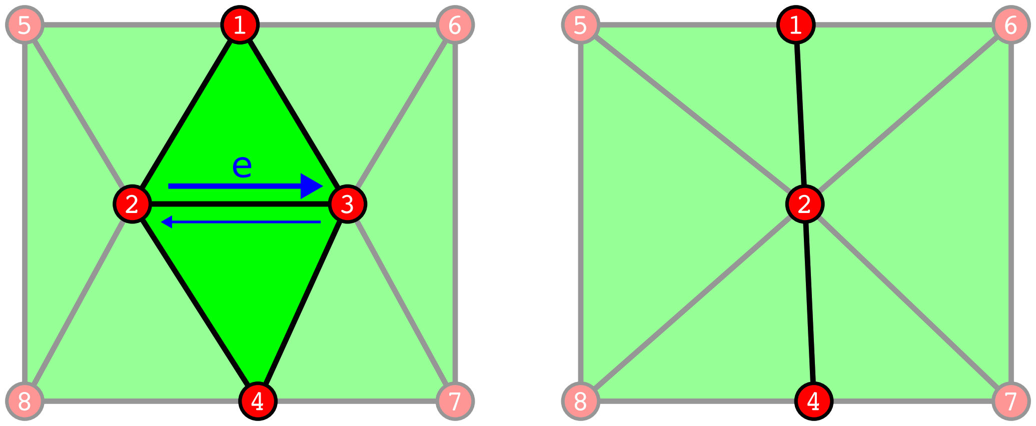


Figure 2.6: Calling the *edge collapse* operation on edge e will cause the two connected faces to collapse and merge vertices 2 and 3.

2.2.2.5 Edge Flip

The *edge flip* operation can be used to change the tessellation of two adjacent triangles that comprise four unique vertices. These vertices can be considered as the corners of a quad. A quad can be triangulated in two different ways, using one of the diagonals as the separating edge. The edge that is supposed to be flipped is one of these diagonals. Flipping it will change it from one diagonal to the other.

The diagonal edge and its mate have to get different vertices, turning them into the second diagonal. Finally the two faces have to be reconstructed. This operation can be called when optimising the mesh. It can result in a more regular triangulation and thereby improve the quality.

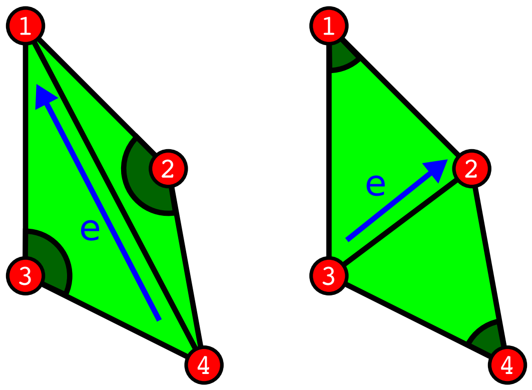


Figure 2.7: The sum of the opposite angles at the vertices 2 and 3 is larger than 180 degrees and the so called Delaunay condition is not met. The flip operation transforms the triangulation into a valid Delaunay triangulation.

2.2.3 Automatic Vertex Normal Generation

In OpenGL normal vectors are used for the per vertex lighting calculations. The face's *normal vector* can easily be generated using the cross product of two edges. A single normal vector per face is sufficient for *flat shading*. The problem with flat shading is that there is no smooth transition between two faces. This leads to clearly visible edges and makes the object look even more artificial.

To enhance the lighting quality, *smooth shading* has to be activated. This mode requires one normal

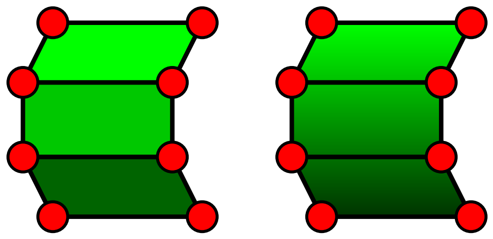


Figure 2.8: Flat shading on the left versus smooth shading on the right side

vector per vertex. These normal vectors can be calculated by interpolating between all the adjacent faces' normal vectors. The problem with this general approach is that there will be no edges visible at all. Depending on the mesh, this might not be what it is supposed to look like. A cube without clearly visible edges is a rather strange looking object. A few details have to be added to the calculation in order to cope with these problems. An automatic measure would be to add a certain threshold. Only face normal vectors that are pointing away less than x degrees should be used for interpolation. This would solve the cube problem, for instance, but in some cases it might not be correct, depending on the threshold. One solution to this problem is to set some edges explicitly sharp. Usually the modelling tools know best which edges are supposed to be sharp and which ones are smooth.

One vertex can be used by multiple edges. If one of these edges is set to sharp, this implies that the two faces connected by this specific edge also share a common vertex. However, they only share the vertex position and not the vertex normal. There can be more normal vectors than vertices. They need to be stored on a per-half-edge basis.

As long as a face is not modified, the normal vectors will stay the same. But as soon as it is modified, the normal vectors of all connected vertices must be recalculated.

2.2.4 Texture and Colour

To improve the visual quality of a mesh it should not just be single-coloured but also have different appearances, depending on its material and properties. OpenGL provides several tools to colourise a mesh. The most common tools put a texture on top of a face, sometimes several of them simultaneously, or defining a colour for every vertex and letting the GPU do the interpolation. The quality of the texture method can easily be improved by replacing the texture file, whereas the vertex method relies on the mesh itself and the amount of vertices.
To define which part of a texture is drawn onto a face, each vertex needs a set of *texture coordinates*. Providing proper texture coordinates for every vertex in an arbitrary mesh can be a non-trivial task. Unwrapping a full mesh onto a single texture and placing each face in a separate area is a time consuming operation. That is not even the biggest problem of all. When modifying a mesh, some faces will grow and others might disappear completely. The faces' areas have changed, but in the texture they still take up their old areas. Faces with distorted appearances will be the result. Especially when adding new faces, the unwrapping procedure will have to be run again, making the contents of the old texture unusable. It is almost impossible to transfer the old contents to the new texture without at least losing some details.
That is why the texture is only used in a *detail texture* kind of way. The same texture repeats itself multiple times over the whole mesh and only provides a certain surface structure. If new geometry is added to the mesh, the texture coordinates can be extended. Hence the existing texturing remains unchanged. By doing this the texture can be free from distortions. However, if the mesh is deformed rather than extended, distortions will occur sooner or later. They will only be less noticable.
In order to colourise a mesh, colours are assigned to the vertices. Depending on the individual polygon size this usually is an adequate way to solve this problem without adding too much complexity.

2.3 User Interface

2.3.1 Environment

Interacting with the user is a very important element of an application. Providing the means to easily perform various actions can be a challenging task. In order to fully utilise the potential of a DAVE-like environment, different aspects have to be considered. The environment generated by the programme has to be both good looking and functional at the same time. Therefore every part of it has to serve a purpose. For example, it will seem awkward if there is something that resembles a button, that is in fact something else. The purpose of every object has to be clear immediately without further instructions.

The user can move around freely in the DAVE. As the area however is rather limited, the space has to be used efficiently. Most of the area is taken up by the object or drawing the user is currently working on. This object is located at the centre, hence controls and other parts of the user interface have to be placed somewhere else. In order to make it seem more immersive, the virtual environment has to be of the same size as the real environment. Moving one step to the left has to result in an appropriate action in the virtual world. The real and virtual floors therefore coincide. By limiting the area, a user will not try to reach places that are too far away. The danger of the user running into one of the DAVE's walls is thereby greatly reduced. Putting virtual objects on the floor is not recommended as the user can stand on, or actually in them. Hence all control elements have to be placed on the sides, but still within the user's grasp.

As the user is able to change parameters and select different tools, some way of displaying information should be added. For example, the radius parameter can be visualised by drawing the cursor in the right size. In addition, a description of the currently selected tool, or maybe a few hints here and there can be very useful. This information can either be displayed next to the input device or at a designated message area. The advantage of showing messages next to the input device is that the user can read them without having to look somewhere else. However, displaying this information can occlude objects that lie behind it. The position of the input device can vary due to the optical tracking system's lack of precision. Hence the information might appear to *fly* around. As a result, a fixed message area is responsible for displaying these messages. It is located outside of the drawing / sculpting area and will not occlude important parts of the scene. Letters must be easy to read from any direction and distance and therefore it seems reasonable to reserve an adequate area. As the DAVE only has three walls, the front wall seems to be the most suitable one for this application purpose.

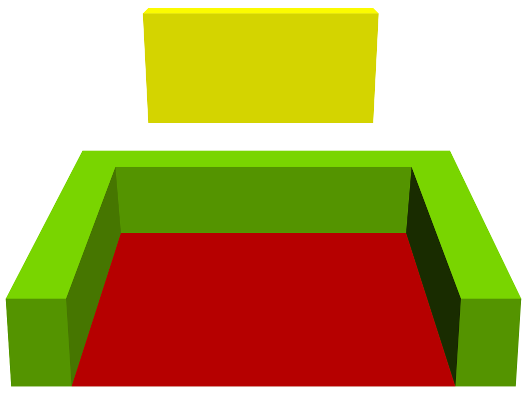


Figure 2.9: Sketch showing the basic layout of the virtual environment. The walkable area, coloured in red, coincides with the real floor. The yellow rectangle symbolises the message area, whereas the green cuboids are areas that can contain selectable tools.

This layout, despite being very simple, allows great flexibility. Not all of the reserved space has to be used for tools. Depending on the current requirements, tools can either be added or removed.

It will never appear cluttered because only the necessary tools are visible. The remaining area does not have a specific purpose yet. It can be used to show the non-interactive *surroundings* of the environment. If the environment is supposed to be outdoors, this area can consist of a landscape and the accompanying sky. Even though a landscape can be very colourful, it will be obvious that it only serves as a decoration. This is a property that cannot be guaranteed when using a different kind of environment, for example inside a house. Little details like cupboards and shelves can make a house appear more realistic. The apparent danger is that it will distract the user and this should be avoided. As the surroundings can easily be exchanged, different types can be tested.

2.3.2 Info Screen

Displaying information inside the DAVE environment is of great importance. Whether they are instructions or simple debugging messages, there is a need for visualising data. For this reason the environment contains a special area used for exactly this purpose called the *Info Screen*. It should enable the application to display both text and images. As the space is limited, the information has to be presented in a short and uniform way - not every message should look completely different. Hence a special layout has to be devised that takes care of arranging images and messages.

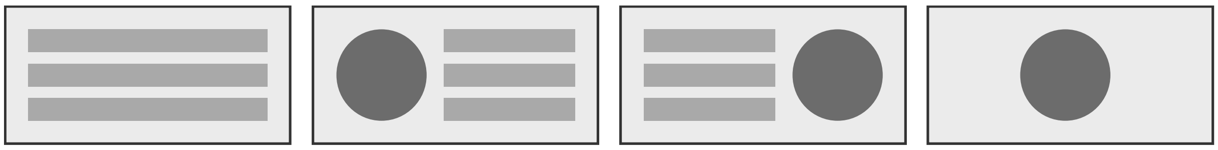


Figure 2.10: There are four different layouts: text only, text and an image on either side, or only an image and no text at all. As the text has to be readable from far away, the font size is rather large. Not more than three separate lines of text can be displayed.

A message usually only has to be displayed for a certain length of time. Once the user has finished reading the message, it is no longer necessary to display it. After a predefined duration, the message starts to fade out. Hence the user can immediately spot new messages as soon as they appear. Some messages might require the user to perform special actions. These messages will not disappear until the user has complied. Moreover, they should not be replaced by other less important messages. Therefore the info screen's contents can be fixed.

2.3.3 Tool Selection

Instead of using complicated combinations of buttons to activate different editing modes, a more intuitive way has to be devised. Very often real and virtual objects are mixed to achieve a more immersive user experience. For example a painter might carry an extra board that turns into a colour selector in the virtual environment. This board can be used for different purposes. By placing virtual buttons on it, the functionality can be increased greatly. In fact, such a device can be turned into almost anything. The only problem is that real objects do not blend in very well in a virtual environment. As the environment is projected onto the walls, it is not possible to change the appearance of the board. There will be no visible feedback on the board if the user presses a button. In addition to that, the DAVE environment itself is rather dark. No light sources are supposed to interfere with the image projectors. Hence reading information or trying to recognise specific controls on a board are very difficult, so this approach does not seem well suited for a DAVE-like environment.
This programme will only require a single input device. Therefore special handling is required in order to distinguish the different editing modes. The device can either be used to manipulate an object or to select different parameters and tools. As long as one mode is active, no functions of the other

mode should be called by accident. Thus unwanted surprises are avoided. Switching between the different modes should be self-explanatory. This can be achieved by using very simple mechanism. A tool in the DAVE environment is represented by a model. The colouring mode for example is symbolised by a brush model. Clicking on the brush will activate the corresponding mode. Which tool gets selected is determined by proximity to the input device. By leaving gaps between the tools this selection becomes unambiguous and the user does not have to be precisely at the right position. When tools are situated on the sides they do not interfere with the object that the user is creating in the centre. In case the user somehow manages to create an object that extends to the sides, *tool selection* will nevertheless be possible. By prioritising the tools over the object, they are always available.

- increase cursor size
- switch to sculpt mode
- switch to paint mode
- save mesh
- decrease cursor size

- snake modeller
- landscape modeller
- blob modeller cube
- blob modeller sphere
- blob modeller cylinder

Figure 2.11: One possible tool selection layout. The different types of modelling tools on the right are separated from editing tools on the left.

2.3.4 Cursor

Being able to modify an object is an integral part of the programme. A user can however only interact with an object if it is obvious which part is currently being modified. Hence the selected part has to be highlighted in a way that neither completely occludes the object nor is too ambiguous. The visualisation can be regarded as a *cursor*. By moving the input device, the selection will change, similar to a mouse influencing the position of a pointer on an ordinary computer. The cursor's position is calculated by putting a straight line through the shutter glasses and the joystick. The optical tracking system provides the necessary information and the resulting direction can be used to select the part of an object the user is currently pointing at.

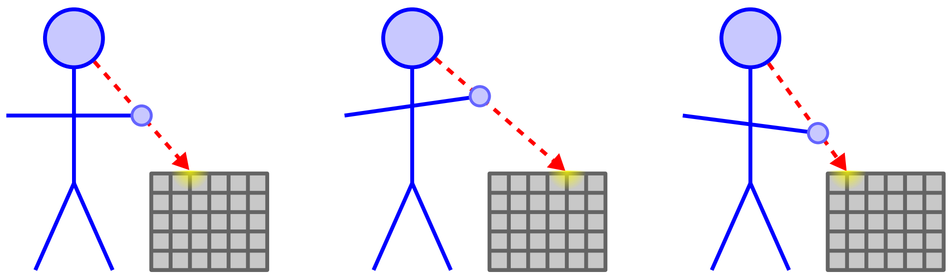


Figure 2.12: Diagram depicting the eye-joystick relationship. By moving the joystick, the selection changes. The selected part of the object always appears to be right behind the joystick from the user's point of view.

This way of determining the selection relies only on the position of the joystick. However, the tracking system also provides the absolute rotation. An alternative way of calculating the position of the cursor could incorporate this rotation. Instead of creating a straight line originating from the shutter glasses, the line could originate from the joystick. The direction would then be based on the rotation of the joystick. Hence, the joystick can be rotated as well as translated in order to change the position of the cursor. This method is more powerful than the previously described eye-joystick relationship as more possible cursor positions can be selected from a single location. However, as the calculations now depend on two different parameters provided by the optical tracking system, the lack of precision will be more noticable.

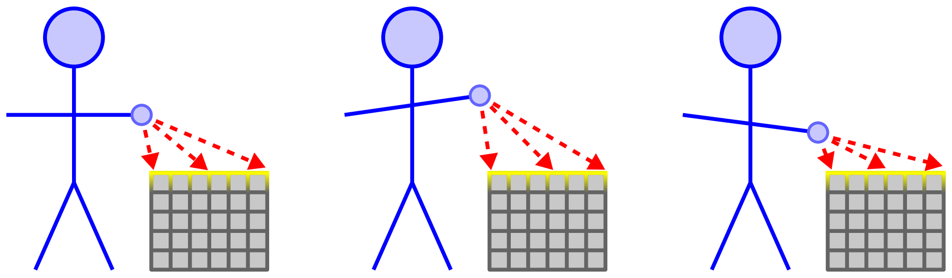


Figure 2.13: By moving and rotating the joystick, the selection changes. In this example the rotation of the joystick has more influence than its position as the same points can be selected each time.

In addition to the position, the cursor has a size parameter that can be regarded as a radius. Vertices that lie within this radius are influenced. Hence the selection is spherical. Visualisation on a flat surface corresponds to a disk. Projecting any kind of colour or texture onto an arbitrary surface can be rather complex. This problem can be solved by utilising the capabilities of modern graphics hardware. A procedural texture shader [RLK09] can calculate the distance to the cursor's position for each point. Depending on this distance a point is either part of the selection or not. Therefore it is possible to draw the precise selection onto a mesh. The *OpenGL Shading Language* (GLSL) can either access and modify vertices using a vertex shader or change the mesh's appearance by altering the pixels using a fragment shader (pixel shader). In this case the fragment shader will do most of the work as the vertices will not move.

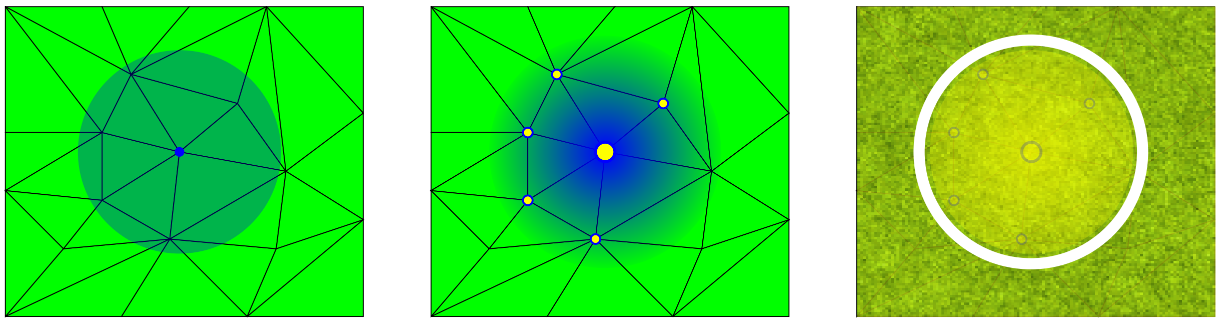


Figure 2.14: The cursor is defined by a position and the radius parameter. The position coincides with one specific vertex. Based on this position, the surrounding vertices are selected. The influence on each vertex depends on the distance to the cursor's position. Hence near vertices will be influenced more compared to vertices that lie on the cursor's border. Instead of showing the individual vertices that form the selection, the shader programme will draw a semi-transparent circle on the mesh's surface. A gradient depicts the cursor's influence. As the surface could be of the same colour, an additional white circle is drawn. Hence the selection will be visible on any surface. In case the user is performing an action and the selection is fixed, the yellow part can become more transparent. Therefore changes to the mesh are visible at any time.

2.4 Modelling Tools

2.4.1 Pipes

The *pipe modelling tool* can demonstrate the *sketching* capabilities. The first thing a user will try to do in the DAVE environment is to create an object by drawing multiple strokes with the input device, much like with a pencil on paper. Instead of having flat ribbons floating in 3D space this modelling tool will create pipes that can be viewed from any direction. The user can freely choose the diameter and create either very thick or rather thin pipes. A pipe consists of connected cylindrical segments. As a pipe gets extended, the number of segments increases.

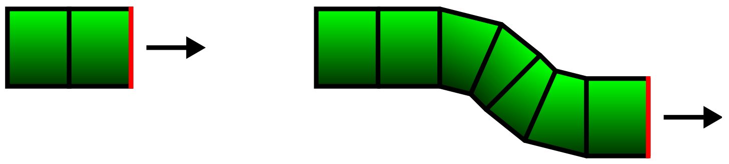


Figure 2.15: A pipe can be extended in either direction. Once the appropriate end is selected (depicted with the red line), the user can start to draw. As the input device is moving, more and more segments are added to the pipe.

The user can add new pipes or extend the existing pipes. Which of these two actions is performed is determined by the stroke's starting point. If this point is nowhere near the other pipes, a new and independent pipe is created. To extend a pipe, the starting point will have to be close to an end of a particular pipe. Therefore the modelling tool will have to keep track of all ends. This enables the tool to quickly look for the nearest end. If the distance is lower than a certain threshold, the end gets selected. A reasonable value for the threshold will have to be determined by conducting tests. If it is too large it will be difficult to create pipes that are close to one another, whereas a small value will make it almost impossible to select a specific end. In case this way of drawing proves to be troublesome, these two actions might have to be separated completely. This very basic way of drawing pipes can be extended by more advanced features:

- delete segments and thereby shorten pipes
- delete whole pipes
- connect two ends and merge pipes
- cut a pipe in two
- alter a pipe's radius later on
- change shape of a pipe after it was created

None of these features are required at the beginning. Adding them is considered optional as they are not going to be needed by a casual user. Drawing pipes can be done by pressing a single key, whereas the more advanced features have to be activated in a different way, like for instance by using the tool selector.
In order to avoid unpleasant artifacts, pipes are not permitted to have sharp corners. Smooth bends are preferred and are therefore enforced during the creation process. Usually a segment's length is rather short. This enables a pipe to quickly adapt to new directions while retaining its smoothness.

Besides the geometry aspect, colouring the pipes is another important feature as it can change the whole appearance. When adding new pipes, they will be using the currently active colour. Hence the right colour should be selected before creating the pipe. If parts of a pipe are supposed to have

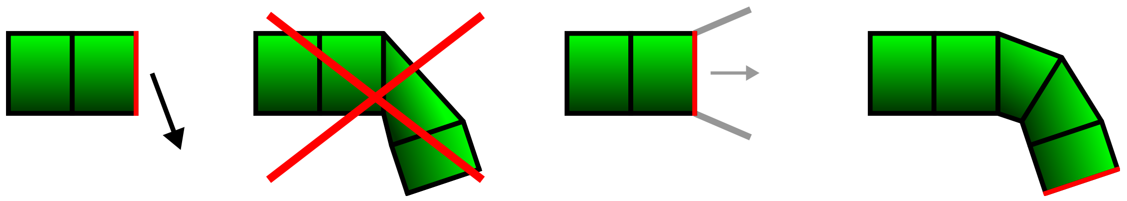


Figure 2.16: In this little example the user wishes to abruptly change the direction. Without any further handling a sharp corner would be created. By defining a maximum difference in the direction between two consecutive segments (represented by the grey lines in the third image), the new direction can slowly be attained.

a different colour, the user only has to select the brush tool and apply it. The brush will have to be held over the area of interest and the changes will be visible instantly. This should be a very intuitive way of interacting with the objects. By default, everything that lies within a certain radius will be coloured. However, if a second pipe is near enough, it will also change colour. In case this proves to be too annoying, a primary pipe will have to be selected.

2.4.2 Blobs

Drawing pipes in 3D space can be an interesting experience. Nevertheless it does not fully utilise the mesh data structure's capabilities as it focuses on sketching rather than modelling. The *blob modelling tool* therefore provides the means to transform an arbitrary object into a sculpture. As the process of turning an object into the desired shape can be challenging, not all modelling attempts will be successful. The results can often be undefinable, hence the designation *blobs*.
The user can choose from a number of predefined shapes, preferably the one that is closest to the target shape. By picking the right shape, the amount of work is therefore reduced:

- Sphere
- Cube
- Cylinder
- Cone
- Pyramid

These are just a few examples. More shapes can be added on demand. Models that were created with external tools can be imported and used as templates as well.
A mesh can be deformed by selecting a part and moving it. This part is defined by selecting one specific vertex, usually the one that is closest to the input device. One of the features of the mesh data structure is that it provides fairly regular triangulation. If the user tries to select a specific point on a surface, there will always be a vertex nearby. Depending on the currently set parameters, adjacent vertices become part of the selection. The distances between the original vertex and the other selected vertices are stored. The deformation is applied to the selection by using a Gaussian curve. Vertices near to the original vertex will be moved more than vertices that are further away. This will create smooth bumps instead of sharp edges and reduces the amount of optimisations to be performed later on.

The modelling tool provides two different editing modes. A user can either select a part of the mesh and drag it to the right position, or perform a more *sculpting* like action. The first mode keeps a certain selection until the editing operation is completed whereas the sculpting mode continuously updates the selection. Sculpting displaces the selection by a fixed amount, which is supposed to

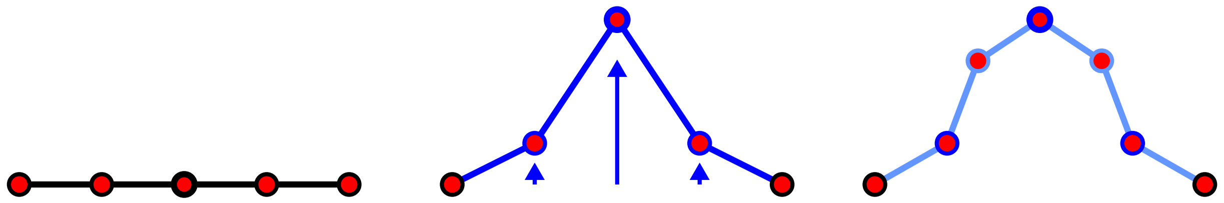


Figure 2.17: The middle vertex is selected. Depending on the current cursor radius, the adjacent vertices are influenced. When pulling the selected vertex up, the neighbours will therefore move as well. As soon as the translation is complete, a smoothing operation is executed. In this case new vertices are added to the mesh.

resemble sculpting in real life where parts of an objects are removed with hammer and chisel. The advantage of virtual sculpting is that the modelling tool can add new parts again if too much was removed.
An important aspect of this modelling tool is that it must determine whether the original shape of the mesh should be preserved or not. A shape is usually defined by sharp edges. Depending on the user's preference, these edges can either be retained or modified at will. Therefore this aspect will require special handling prior to the editing operation as well as afterwards. By default the user can modify any parts of a mesh and thereby destroy sharp edges. As the sculpting tools are designed to create bumps and dents, the resulting objects are likely to be organic rather than man-made.
As soon as the object has the right shape, the user can start to change the appearance by colourising it. An arbitrary sculpture can get a whole new meaning if the colouring is modified. A sphere, for example, can change from resembling a planet to being a head in no time. Just by adding a few well placed spots the sphere can get eyes and a mouth.

2.4.3 Landscapes

Creating and visualising a terrain with computer graphics is a very common task. One way of implementing this task is to use *heightmaps*. For every set of coordinates on a plane a height value is defined. Depending on this value a point will either be moved up or down. Heightmaps can be stored in an image and therefore be modified with any raster graphics editor. The degree of realism depends on the image's resolution. By increasing the amount of pixels, smaller details can be added to the terrain. This makes it possible to either store only the most important hills and features, or almost every little bump. Nevertheless this approach has certain limitations. As all points are displaced along the same axis, it is not possible to represent caves or features that require points to be moved into other directions. Hence heightmap rendering is sometimes called *2,5D* instead of 3D. There can never be one point on top of another one, as the x and z coordinates remain unchanged.

Flight simulators use heightmap rendering technique to visualise large terrains. As the geometry's complexity can get very high, certain optimisations are necessary. Mark Duchaineau et al [DWS*97] presented a way of adapting the geometry in real-time. Depending on the distance to the viewer, the mesh quality can be altered. If a hill is far away, the number of triangles can be reduced without the viewer noticing it. This approach made sure that there was no visible gap between different levels of detail. Maintaining a continuous triangulation is a very complicated task. As landscapes created with this modelling application are comparatively small, it will work without these optimisations. Hence other aspects can receive more attention.
The *landscape modelling tool* is very similar to the blob modelling tool in respect to the requirements of the software. A user should be able to grab an area of the mesh and either push or pull it in a direction. In this case the direction is fixed, resulting in a heightmap like *2,5D* structure. Using this structure there are two different techniques for manipulating the terrain that can be implemented.

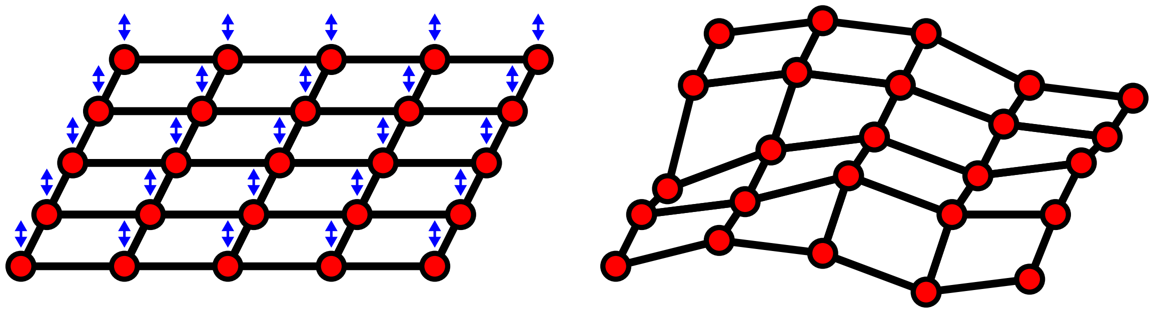


Figure 2.18: Simple example showing the basic functionality of a heightmap. Only the height values are changed in order to transform a flat surface into something that resembles a terrain.

The user can either grab an area and turn it into a hill by pulling the input device up, or perform a continuous operation. Such a continuous operation will constantly increase or decrease the area that is near the input device by a small value. This way the user can freely move around while editing the terrain. The grabbing method can quickly create large hills, whereas continuous manipulation provides a very comfortable way of editing the terrain. Once the user has figured out how to use these methods, a flat plane can be turned into an arbitrary terrain in no time.
Colouring a terrain will work out of the box as this function can be reused from the blob modelling tool. However, only a limited number of colours is appropriate for landscapes. Pure red for example will seem unnatural and should not be applied. Therefore the palette can either be limited to a number of suitable colours or the whole colouring can serve a different purpose. One way of utilising the data would be to apply multiple textures instead of plain colours. The red, green, blue and alpha channels can each represent different textures. If a colour is red, the first texture will be used. Green stands for the second texture and so forth. Hence no further information has to be stored, only the rendering process will have to be adapted. If this method proves to be successful, the other modelling tools might take advantage of it as well.

Chapter 3

Implementation

3.1 Software Requirements

The DAVE's computers can be run with *Linux* as well as *Windows*. Originally, Linux was the prime operating system. Nowadays, Windows has become just as important. Hence the software has to be capable of running in a either environment. The *libraries* used by the software have to be picked accordingly. As many libraries are compatible with both Linux and Windows, the final application will be able to run in the DAVE, no matter which operating system is currently active. There are two common ways of implementing applications for the DAVE: *OpenSG* [RVB02] or *Davelib* [FHH03] based. OpenSG is a scene graph library that is capable of many advanced features like multithreading or special rendering techniques. Additionally it enables applications to be synchronised via network, which is necessary in this particular multi-projector environment. The alternative is called Davelib and was developed for the first DAVE in Braunschweig. Utilising the Davelib, normal OpenGL based applications can be adapted to run in the DAVE environment, allowing multiple instances of the same application to be synchronised. The popular Tux Racer game for example became a DAVE demo application by incorporating the davelib. Besides the ability to synchronise, the Davelib provides other important features such as parsing configuration files, requesting data from the tracking server and calculating the *projection matrix*, including projector calibration. This matrix enables a seamless projection, as even the slightest projector alignment error can be corrected.
As the modelling application will use a custom mesh data structure, the OpenSG library seems almost unnecessary. Conversions between this custom data structure and the common OpenSG mesh representation will probably have adverse effects on the performance and add unwanted complexity. Hence the Davelib seems to be the more reasonable choice. Not all parts of interest are covered by either Davelib or OpenSG. Therefore extra libraries have to be added:

- **Davelib**: configuration, communication with tracking server, UDP synchronisation, projection matrix
- **SDL**: window management, input handling
- **SDL_image**: loading of arbitrary image files like PNG or JPEG
- **SDL_net**: TCP communication
- **OpenGL**: 3D graphics
- **OpenAL**: 3D sound

Davelib originally used OpenGL Utility Toolkit (*GLUT*) instead of Simple DirectMedia Layer (*SDL*). Therefore the modelling application used GLUT at the beginning. However, placing windows

at specific positions with fixed sizes was troublesome and caused some problems. As SDL proved to be reliable in previous applications, GLUT was eventually replaced by SDL. The only disadvantage of SDL is the lack of multi-window support. Future SDL versions will contain this feature. Hence this part of the Davelib is currently unavailable.
The SDL library has been used in countless applications, especially games. It enables programmers to create platform independent applications. However, it only contains basic functionality. More advanced features can be found in separate libraries. While SDL only has the ability to load bitmaps, SDL_image can load virtually any image, which makes it a perfect addition. SDL_net is, as the name suggests, responsible for the network part. Even though it is capable of both TCP and UDP, only TCP will be needed. UDP functionality is already covered by the Davelib.
The only aspect that still needs attention is sound. There is a SDL library designed specifically for this purpose. It is called SDL_mixer. However, it is not capable of creating surround sound. Therefore the Open Audio Library (*OpenAL*) is used instead. This library's application programming interface (API) [Hie06] is very similar to the one provided by OpenGL. Hence if a programmer is familiar with OpenGL, using OpenAL will be comparatively easy to learn. In general, OpenAL consists of three different kinds of objects: audio buffers, sources and listeners. Sound files are stored in audio buffers. Sources can play these buffers back. By defining properties like position and speed for sources and listeners, OpenAL can calculate the sound intensity and apply the Doppler effect for instance.
All SDL libraries as well as OpenAL are distributed under the lesser general public licence (LGPL) version 2.

3.2 Tracking Simulator

3.2.1 Overview

It is not always possible to use the DAVE, and when trying to develop new features it might be easier to test everything on one computer instead of having to put on the shutter glasses, grab the joystick etc. That is why the *Tracking Simulator* was created. It should provide a convenient way of moving an object in a virtual DAVE environment without actually having to be there.
There already was a tracking simulator available, but it is difficult to perform complex movements with it necessary to test a mesh creation programme. A new simulator now exists which is able to move both the camera and the joystick simultaneously by only pressing a few keys. With this new simulator it is now possible to easily recreate movements because a single test is usually not enough and the same operations have to be performed several times. By pressing the shift-key the movement speed is increased, whereas the control-key reduces the speed to allow more precise movements.
When executing the programme the user sees a simplified version of a DAVE environment, showing the walls, the camera and joystick markers.

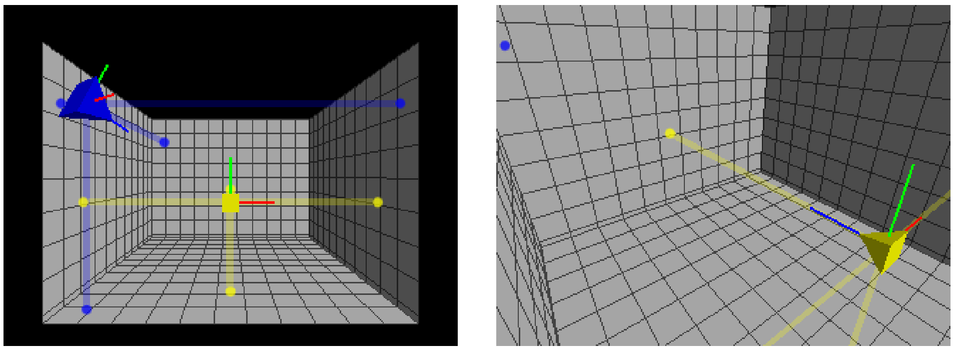


Figure 3.1: Screenshots of the Tracking Simulator. The left picture demonstrates the default overview mode, whereas the right picture shows the scene from the camera's point of view.

3.2.2 Tracking Protocol

The protocol used by the tracking server, and also the simulator, is ASCII based. As soon as a request is received, the positions are transmitted. Such a reply consists of the following parts:

- Length of the reply in bytes
- Optional packet ID from the request
- Either *MarkerPos* or *Target*, depending on the type
- Accompanying data

A single reply can have multiple markers and targets. Usually a client will query all available data with a single request. A marker's or target's data is formatted in a particular way using commas and brackets, allowing an easy interpretation on the client side.

3.2.3 Joystick Emulation

In the real DAVE environment input events like joystick buttons are transmitted via a wireless connection to the master application. These events are not part of the tracking protocol itself and have to be processed separately. When using the simulator the user's hands are usually busy pressing several keys in order to move objects. That is why there is no hand available to hold a joystick. The joystick itself should be simulated as well.
In a normal desktop environment, only the active window will receive input events when the user is pressing a key. The simulated joystick buttons should be part of the same application the user is working with in the first place. One way of transmitting data to the master application would be to create a new connection with a separate protocol which would unduly complicate the process. There already is a connection between the simulator and the master application. Adding extra functionality should not interfere with its original purpose and leave the tracking protocol unchanged.
The solution to this problem was to add extra markers with a special behaviour. A marker's position can be used to store up to three buttons. The x coordinate, for example, can be associated with the first button. A value equal to zero would indicate that the button is not being pressed at the moment. Using this technique an infinite number of joystick buttons can be simulated. The protocol might not be designed to handle situations like these, but as markers are identified by their names, programmes only pick the ones of interest and discard the rest.
The master application now only has to look out for those specially named markers and interpret them accordingly.

3.3 DAVE Simulation

3.3.1 Launch Script

Performing a local test requires the user to start multiple programmes with varying arguments. To speed up the process and reduce the amount of work the user has to do by hand, a special shell script was devised supporting different modes.

- **default**: single application using camera's orientation
- **full**: four instances each showing one of the DAVE's walls
- **ext *address***: only start four client instances and connect to *address*
- **gdb**: debugging with GNU Debugger [SPS02]

- **valgrind**: memory debugging with Valgrind [NS03]
- **dave**: special mode using DAVE settings

All modes except from *dave* and *ext* launch the Tracking Simulator and use a custom configuration file for local testing. The default mode shows the scene from the user's perspective and is therefore exactly what one would see when standing inside the real DAVE. All the other modes serve special purposes. The full mode can check if all programme instances are running synchronously, whereas gdb and valgrind can help to detect programme errors. Valgrind uses its *memcheck* tool to examine all memory related operations. Hence common errors such as accessing uninitialised memory and creating memory leaks can be detected. However, these checks slow down the application considerably and make it almost impossible to use in realtime.
The dave mode can only be used on a computer in the DAVE laboratory because it connects to the real tracking server and input devices without actually using the projectors. Hence important parts of the application such as accessing and processing input data can be tested from outside the DAVE. The ext mode requires an additional address parameter. This address specifies the location of the tracking server and master application. The mode requires a second computer that either started the default, gdb or valgrind mode. Applications can therefore be split up between two different computers. One of them is responsible for all the input, the second one only uses clients to create output. This mode is particularly useful if screen space or computational power is limited. Instead of having countless small windows on a single screen, only a limited number of windows are left that can be enlarged.

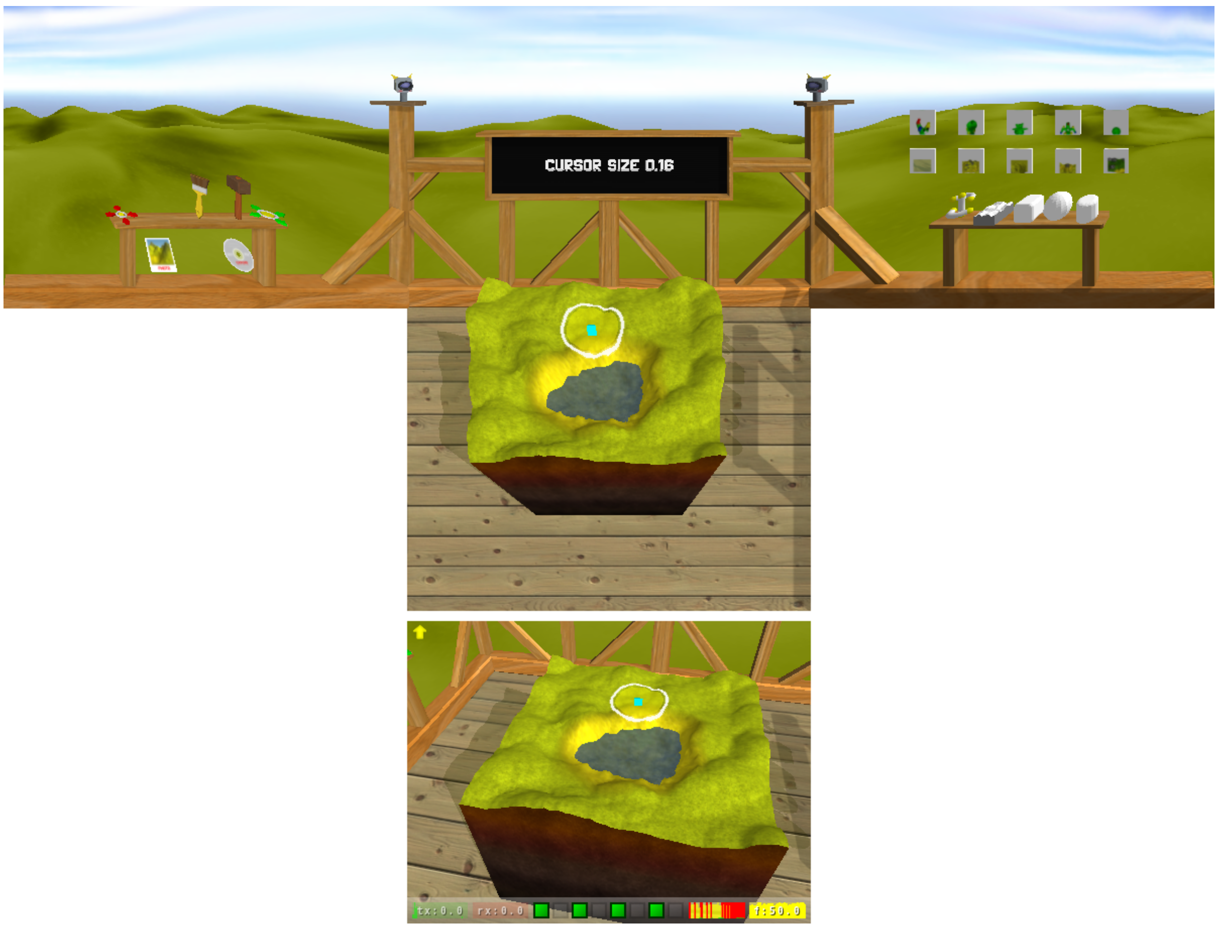


Figure 3.2: *DAVE simulation* using *full* mode. The image on the bottom shows the master application from the user's point of view. The remaining four applications represent the DAVE's three walls and the floor. The images are distorted in order to produce a realistic environment from the user's point of view. By removing the window decoration, the applications merge to one seamlessly connected image. Each window is running at a resolution of 640x400 pixels with the *double_size* configuration parameter activated. The rendering process is rather time-consuming, the framerate is as low as 50 per second. Therefore the draw load graph, the second graph from right, is signalising a too long draw time, recognisable by the red colour. The computer that was used to create this screenshot simply was not fast enough. Nevertheless, the applications were still quite responsive.

3.3.2 Limitations

Simulations definitely help to speed up the development process, but from time to time *real* tests are inevitable. Tasks will be performed in a different way. For example when letting other people try out an application they do not know. It will help to detect errors and point out shortcomings in the user interface. Developers already know how to operate the application. Buttons, for example, will be pressed in the *right* order. Due to the fact that this order seems obvious for the developer, no further improvements are made. Hence certain use cases might be overlooked.
One aspect that becomes immediately apparent is that everything seems to be much larger in the DAVE when comparing it to the simulations. A mesh that fills the computer's screen still seems comparatively small, whereas in the DAVE the same mesh is a few meters wide and thus fills the whole available space, leaving the user nowhere to stand but inside of the mesh. This could never happen

with a real object, hence it just seems wrong. Objects and tool settings usually have to be scaled down. If, for example, a single pipe already blocks the user's view and prevents the creation of more pipes, it is obviously too large. Errors like this can spoil the first impression a user gets.

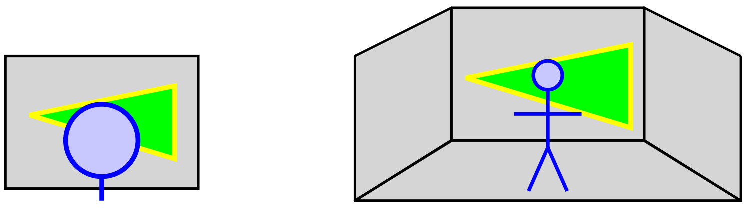


Figure 3.3: If a triangle is taking up the entire screen of a normal monitor it can turn to be as large as the user in the DAVE.

A full simulation requires a great deal of computational power. If all graphical details like for instance shadows are activated, the performance can become an issue. The graphics hardware is the bottleneck. Hence certain features can be deactivated in the configuration file. Low performance hardware found in netbooks requires the deactivation of all non-essential features. These features are:

- draw environment
- draw shadows
- toggle shader programmes
- reduce window size

By altering these settings, the performance can be improved without reducing the functionality. Deactivating the shader programmes usually has a significant influence. If, however, the application is still not running at a reasonable speed, the mesh quality parameter can be adjusted.

3.4 Input Handling

3.4.1 Joystick

Accessing joysticks is part of the SDL. Receiving button events is very easy to implement. However, there are several joysticks available in the DAVE. They all share a common name and have identical properties. They cannot be distinguished. In addition to this, their order can change in case joysticks are added. Hence special handling is required to identify the joystick of interest. This joystick is determined by examining the first button event. The sender identification is stored. From then on all events sent from other joysticks are discarded. Therefore only the remaining joystick has influence on the application.
The *joystick* has a rather unique appearance. Parts of a normal joystick and a wireless gamepad were merged and equipped with markers for the tracking system. As the tracking system takes over the movement part, the joystick no longer needs its base. All that is left is the stick. As cables would constantly be in the way, joysticks need to be wireless. Hence the necessary parts were extracted from a wireless gamepad. The joystick now has the ability to send button events directly to the master computer.

1. Trigger
2. Pull out
3. Increase cursor size
4. Push in
5. Decrease cursor size

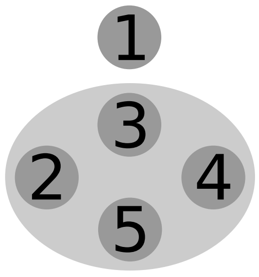


Figure 3.4: The joystick has four different buttons on the top and a special trigger button on the front. The trigger is a special button. Instead of just detecting button events (pressed or released), it measures how far the button is being pressed. As such, it acts as one of the joystick's input axes rather than a button.
The *trigger* is operated using the index finger whereas the remaining four buttons can be pressed by the thumb.

3.4.2 Master Application Interface

The main way of interacting with the application is to stand inside the DAVE and use the joystick. Usually applications are started remotely using a handheld device. Therefore, one does not have to leave the DAVE in order to switch to a different application. The *master application* itself however is running on a computer that is located behind one of the walls. As soon as the modelling application is started, a window appears displaying the scene from the user's point of view. This proved to be convenient, because it would otherwise be impossible to see what is going on inside the DAVE. When using this computer, the joystick cannot be operated. The whole drawing and modelling part depends on the optical tracking system. Hence these functions cannot be accessed from outside the DAVE. Instead of the joystick, a keyboard and a mouse are available. However, trying to emulate a joystick using these input devices does not make sense. It would only interfere with the real joystick. In case a test has to be executed, one can always start a DAVE simulation.
There are several configuration parameters stored in a file. In order for modifications to take effect, the application needs to be restarted. However, not all changes require a full restart. In most cases settings can be altered on-the-fly. There just has to be some sort of user interface that allows direct manipulation of these settings. One possibility would be to enable the user to change these settings from inside the DAVE. As the DAVE user interface however is deliberately simple, adding extra settings would complicate it considerably. The alternative is to devise a user interface specifically for the master application. As mouse and keyboard are available, it can resemble a traditional two dimensional user interface. Widgets such as buttons, checkboxes, sliders and other common elements allow easy manipulation of settings. Besides these interactive parts of a user interface, there can also be passive parts. These passive parts do not require user interaction because they only display information. One possible passive element would be a small image that indicates the status of a client. Just by looking at the image, it becomes immediately obvious if a client is still alive or if an error has occured.

3.4.2.1 GUI toolkit

Adding some sort of *graphical user interface* (GUI) to an interactive application is a task that every developer has to deal with sooner a later. There are many different libraries and toolkits available. Some are very powerful while others only provide very basic functionality. As the master application interface is not supposed to be highly complex and only enable the user to perform simple tasks, a custom toolkit was chosen. This toolkit was written for a different application and could easily be adapted. It allows the interface to be drawn on top of the existing window and therefore does not

require a separate area.
The user interface consists of different objects. Multiple objects can be combined to form more complex GUI elements. Objects that belong together are stored and managed by forms. Hence there can be one form responsible for the main menu, another one for displaying debug information and so forth. In case an object is a *widget* (window gadget), it will respond to user input. Only one widget can be activated at a time. If the user presses a key or moves the mouse, events will be triggered. The corresponding form will detect and process these events. This very simple structure is powerful enough to comply with the requirements. Basically every visible GUI element is based on an object. They all share common features and characteristics. The following list shows the hierarchy of the objects:

- **Object**
  - **Label**: display text
  - **Image**: display image
  - **Graph**: plot a variable over time
  - **Indicator**: display status of a variable

  - **Widget**: basic interactive GUI element
    - **Button**: clickable element
    - **Checkbox**: toggle a variable
    - **Slider**: change the value of a variable by clicking on a horizontal bar
    - **SpinBox**: decrement or increment a variable by pressing left or right
    - **LineEdit**: single-line text box

3.4.2.2 Main Menu

The *main menu* can control differents parts of the application. It can be used to change the current modelling tool as well as certain configuration parameters. The purposes can vary greatly. It was necessary to divide the available options into separate groups, thus making it easier to find specific settings. As the application is supposed to support different window sizes, the main menu has to be able to adapt to any size. However, a small window size limits the number of buttons and other widgets that can be display at the same time. Very often this problem is solved by allowing the user to scroll and thereby increasing the available space. In return, not everything is visible at once. In case of the main menu designed for the master application, the space problem is avoided by placing the elements in separate groups and sometimes sub-groups. At the moment only the most important settings are available. Very many settings only have to be configured once. There is no point in enabling the master application to alter them. The DAVE's computers for example support a certain mesh quality. In some cases a higher quality would be feasible, but the current value proved to be sufficient for both complex and simple meshes. Large parts of the menu therefore serve as a kind of remote control. The right modelling tool can be selected and finished meshes saved without having to walk back to the DAVE.
The current menu structure is:

- Modeller
  - Pipes
  - Landscape
  - Blob
- Reset Mesh

- Save Mesh
- Settings
  - Volume
  - Mute
  - Info Bar
  - Debug Mesh
- Quit

The individual items should be self-explanatory. All items except the settings are buttons that trigger actions. The settings themselves consist of a slider for the sound volume and checkboxes for the remaining three options. Sub-categories like modellers or settings are represented by separate menus. As soon as one of them is activated, the appropriate options appear. By pressing the *Back* button, one can always return to the previous menu. Hence, only parts of interest are visible.

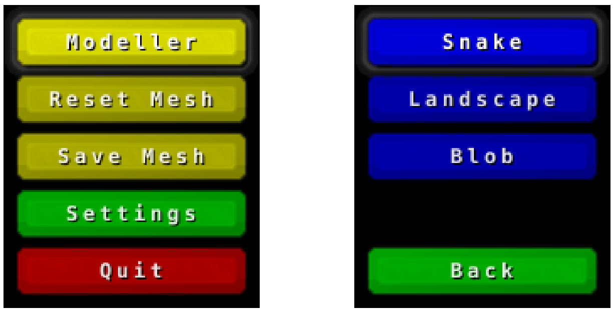


Figure 3.5: Two screenshots depicting the main menu. By selecting the *Modeller* button, a sub menu appears. Different colours immediately show which elements belong together. These menus do not require a large amount of space, hence all elements will be visible even at low screen resolutions.

3.4.2.3 Chat Menu

The *chat menu* provides the means to communicate with the user currently operating the DAVE. In case the user is having a conversation or some noises prevent the user from hearing what one is trying to say, the master application has the ability to send text messages. These messages will be displayed on the info screen and therefore get the user's attention. By giving the letters a distinct colour, it is immediately recognisable when a message is being sent from the master application.

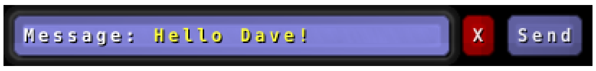


Figure 3.6: Sending a message is a very simple task. The text has to be entered in the appropriate input widget (highlighted in the screenshot). The red *X* button can be used to clear the whole message. If the message is rather long, using this button can be quicker than deleting each character manually.

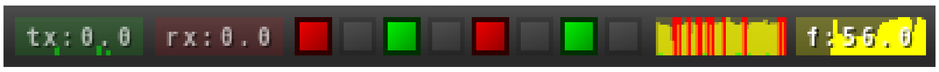
The menu does not contain any further elements. Actually, only the text input widget is required. Message can be sent just by using the keyboard. As soon as the user presses the *Enter* key, the

message will be sent. There is no need to explicitly click on the *send* button. Providing this extra button seems to be common among programmes such as instant messengers.
As soon as the message is sent off, the menu hides again. Hence it is only visible if the user really wants to type in a message. It can be brought up by pressing the '*t*' key. Similar behaviour can be found in computer games. In case the message is supposed to be sent off later, the menu can be hidden by pressing *Escape*. When opening the menu again, the same message reappears.

3.4.2.4 Info Bar

The *info bar* is, as the name suggests, a small bar located at the bottom of the screen. By being semi-transparent, it does not occlude details that lie behind it. Its main purpose is to display information that helps to evaluate the current status of the DAVE environment. Just by looking at the bar, one can immediately examine the current performance and detect errors. As most of the information changes over time, the best way of visualising it is to use graphs. A graph can display the current value as well as its predecessors. Thereby it is possible to determine how the value is changing. This can be very helpful, especially during development. A graph can be ordered to alert the user in case the value reaches a certain threshold by changing its colour or sending a message, if necessary. Even small errors can be detected because they, for example, have an adverse effect on the performance or exponentially increase the number of vertices.
The contents of the bar changed multiple times, depending on the information it was supposed to visualise. As soon as a part of the application behaved as expected, the visualisation no longer provided important information and could therefore be dropped. The remaining widgets mainly serve as a means to evaluate the current performance. A very low frame rate for instance might indicate that the mesh quality parameter is set too high. The current contents is:

- Graph: Network TCP Tx [KB/s]
- Graph: Network TCP Rx [KB/s] (*currently unnecessary*)
- Indicators: DAVE client status: grey = inactive, green = online, red = offline
- Graph: Combined draw and calculate load
- Graph: Frames per second



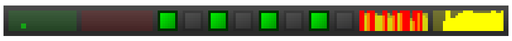


Figure 3.7: Two different screenshots showing the info bar. The bar will try to adapt to the application's window size. In case the width of the window is smaller than 320 pixels, a more compact mode will be activated. In order to maintain a reasonable layout, each object has to be modified in both size and detail. A graph for example no longer displays its caption in order to reduce the required space.

3.4.2.5 Hotkeys

Besides using menus, the application can be controlled by pressing special keys. The functions largely coincide, it can however sometimes be easier to press a single key instead of navigating through a menu. The following list only contains keys not used for accessing menus:

- Quit
- Take screenshot
- Toggle sound
- Toggle mesh debug

All functions, except taking a screenshot, can be found in the main menu. The reason why this function is not part of the menu is that the screenshot should show the mesh, and not the menu.

3.5 Mesh Structure

3.5.1 Supported File Formats

3.5.1.1 Wavefront Object

The *Wavefront Object* (*OBJ*) format is a very common way of storing geometry and other properties of an object. Developed by Wavefront Technologies for their Advanced Visualizer programme it is now supported by many different 3D modelling tools. There is a binary and an ASCII format, latter being the one of interest because of its simplicity.
A file can contain vertices, UV texture coordinates, normal vectors, and faces. Each item is written in a separate line and can be identified by the first characters.

- **v**: vertex
- **vt**: texture coordinate
- **vn**: normal vector
- **f**: face
- #: comment line

The first three types are followed by a list of floating point numbers, whereas the face item is slightly more complex. By using indices, vertices, texture coordinates and normal vectors are referenced. The index corresponds to the absolute list position, starting with one for the first vertex, texture coordinate and normal vector. Indices are separated with a slash character and form a set for each point of a face:
Vertex id followed by texture coordinate id and the normal vector id. Only the vertex is mandatory, the other two can be left blank if they are not specified.
Another approach for referencing items is to use relative instead of absolute list positions, recognisable by negative indices. This approach is not very common and not very many programmes support it.
More advanced features like smoothing groups or free-form curves/surfaces are of no particular interest for this application and are not supported.
The following example shows a basic cube with eight vertices and six faces, created with the latest Blender version. The UV texture mapping is omitted for the sake of simplicity.

```
# Blender3D v249 OBJ File:
# www.blender3d.org
mtllib test.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
```

```
v 1.000000 1.000000 −1.000000
v 1.000000 1.000000 1.000000
v −1.000000 1.000000 1.000000
v −1.000000 1.000000 −1.000000
vn −0.000000 −1.000000 0.000000
vn 0.000000 1.000000 −0.000000
vn 1.000000 0.000000 0.000000
vn −0.000000 −0.000000 1.000000
vn −1.000000 −0.000000 −0.000000
vn 0.000000 0.000000 −1.000000
usemtl Material
s off
f 1//1 2//1 3//1 4//1
f 5//2 8//2 7//2 6//2
f 1//3 5//3 6//3 2//3
f 2//4 6//4 7//4 3//4
f 3//5 7//5 8//5 4//5
f 5//6 1//6 4//6 8//6
```

The *s off* line just turns off the smoothing group and can be ignored.
A full specification can be found online written by Paul Bourke [Boub].

Material Library

Material properties such as colour or texture filename have to be specified in an extra file called the *material library* (MTL). Such a library can contain multiple materials, each of them with a unique name. An object has two parameters to specify which material is supposed to be used:

- **mtllib**: material library filename
- **usemtl**: chosen material name

A new material is added with the *newmtl* parameter and the name, followed by a list of items specifying its properties like ambient (Ka), diffuse (Kd), and specular colour (Ks).
The full specification can be found online [RRT95]. This example shows the corresponding material library for the previous cube example:

```
# Blender3D MTL File:
# Material Count: 1
newmtl Material
Ns 96.078431
Ka 0.000000 0.000000 0.000000
Kd 0.640000 0.640000 0.640000
Ks 0.500000 0.500000 0.500000
Ni 1.000000
d 1.000000
illum 2
```

3.5.1.2 Polygon File Format

The Wavefront Object file format has one major shortcoming: the commonly used OBJ format without any extension does not have the ability to store separate colours for all the vertices. As the half-edge based data structure uses vertex colours, there is a need for a better alternative. The *Polygon File Format* (*PLY*), also known as the *Stanford Triangle Format*, provides this feature. Much like the Wavefront format, there is a binary and an ASCII representation. Again, the ASCII format is the one of interest. Instead of storing all of the properties of the vertices like position and normal vector separately, they are now combined into vertex elements. The structure of such an element has to be defined in the header part. Common vertex properties are:

- **x, y, z**: position
- **nx, ny, nz**: normal vector
- **s, t**: texture coordinates
- **red, green, blue**: colour

Each property can either be a floating point number or an integer. The format has to be specified accordingly. A face only has to reference the vertex elements by their indices.
The header consists of the following parts:

- **ply**: identifies Polygon File Format
- **format ascii 1.0**: selects ascii format
- **comment**: optional comment line
- **element vertex *num***: beginning of the vertex properties, including the total amount of vertices
- **element face *num***: beginning of face properties, including the amount of faces
- **end_header**: end of header

The first two lines are mandatory and define the file's format. The element vertex and face lines are followed by their properties, each of them in a separate line. The header is succeeded by the data. Each line of this data part represents an element. The element's properties are separated by white spaces.
A full specification written by Paul Bourke can be found online [Boua].
The cube example in PLY format is slighty longer than the same mesh in OBJ format. There still are six faces, but as the cube has sharp edges, the faces' vertices have different normal vectors leading to a total number of 24 vertex elements.

```
ply
format ascii 1.0
comment Created by Blender3D 249 – www.blender.org
element vertex 24
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 6
property list uchar uint vertex_indices
end_header
1.000000 1.000000 −1.000000 0.000000 0.000000 −1.000000
1.000000 −1.000000 −1.000000 0.000000 0.000000 −1.000000
−1.000000 −1.000000 −1.000000 0.000000 0.000000 −1.000000
−1.000000 1.000000 −1.000000 0.000000 0.000000 −1.000000
1.000000 1.000000 1.000000 0.000000 −0.000000 1.000000
−1.000000 1.000000 1.000000 0.000000 −0.000000 1.000000
−1.000000 −1.000000 1.000000 0.000000 −0.000000 1.000000
1.000000 −1.000001 1.000000 0.000000 −0.000000 1.000000
1.000000 1.000000 −1.000000 1.000000 −0.000000 0.000000
```

```
1.000000 1.000000 1.000000 1.000000 −0.000000 0.000000
1.000000 −1.000001 1.000000 1.000000 −0.000000 0.000000
1.000000 −1.000000 −1.000000 1.000000 −0.000000 0.000000
1.000000 −1.000000 −1.000000 −0.000000 −1.000000 −0.000000
1.000000 −1.000001 1.000000 −0.000000 −1.000000 −0.000000
−1.000000 −1.000000 1.000000 −0.000000 −1.000000 −0.000000
−1.000000 −1.000000 −1.000000 −0.000000 −1.000000 −0.000000
−1.000000 −1.000000 −1.000000 −1.000000 0.000000 −0.000000
−1.000000 −1.000000 1.000000 −1.000000 0.000000 −0.000000
−1.000000 1.000000 1.000000 −1.000000 0.000000 −0.000000
−1.000000 1.000000 −1.000000 −1.000000 0.000000 −0.000000
1.000000 1.000000 1.000000 0.000000 1.000000 0.000000
1.000000 1.000000 −1.000000 0.000000 1.000000 0.000000
−1.000000 1.000000 −1.000000 0.000000 1.000000 0.000000
−1.000000 1.000000 1.000000 0.000000 1.000000 0.000000
4 0 1 2 3
4 4 5 6 7
4 8 9 10 11
4 12 13 14 15
4 16 17 18 19
4 20 21 22 23
```

3.5.2 Importing Arbitrary Meshes

When using the blob modelling tool, the initial mesh can be arbitrary. Instead of deforming a sphere, one might want to modify a more complex shape, like a car for instance. Therefore such a mesh has to be imported first. Any mesh stored in either the OBJ or the PLY format can be read, resulting in lists of vertices, edges and faces. Luckily the half-edge based data structure already provides the necessary operations to convert these individual polygons into a fully interconnected mesh. First of all the vertices have to be created, followed by the triangles. These triangles can be added by calling the appropriate *addFace* method for each face, using the corresponding vertices as parameters. The vertices are scaled and therefore the original size does not influence the size of the imported mesh. As a result, all imported meshes appear to be of the same size.
The geometry is now fully converted, only certain vertex properties are still missing. If vertex colours are available, they can be assigned to the new vertices right away. As it was mentioned earlier, texturing a mesh can be a problem. Hence the original mesh's texture coordinates will have to be discarded. Substituting the texture coordinates with *usable* coordinates remains an unsolved problem. The only task left is to call the sharp edge detection operation. This is a very important step as it will help to preserve the original mesh's shape. Once the mesh is fully imported, the half-edge data structure will try to improve the quality by re-tessellating the surface and forcing a more or less uniform edge length. This operation can change the mesh significantly, only the previously detected sharp edges will stay in the same place.

3.5.3 Exporting Meshes

When the user is finished creating an object, it might be worth saving and needs to be exported to a format that almost any 3D modelling programme can read. Just like the import operation, the export operation supports OBJ and PLY files. The PLY file format is preferred, as it can store all the required properties, including vertex colours. The export operation can easily be accomplished by just creating lists of vertices, edges and faces.
As an additional feature, a picture of the mesh is stored in a separate file. Utilising OpenGL's *framebuffer object* technology (FBO), the mesh is rendered to a texture using a predefined camera. The texture is then stored in an image file, alongside the mesh. This makes it easier to identify the

right file without having to load the mesh.
Depending on the quality parameter, the number of triangles can be relatively large. Flat surfaces get divided into smaller triangles. In order to describe the geometry of the mesh, not all triangles are necessary. By decreasing the quality parameter, small triangles are merged. Sharp edges will stay in the same place and preserve the mesh's shape. Fewer vertices result in a reduced quality as far as the colouring is concerned. Therefore the full quality is exported by default.

3.5.4 Quality Parameter

To make sure that the application is running at a reasonable speed, a *quality parameter* can be set for different systems. The parameter influences the number of faces by setting a preferred edge length. A fast computer can support higher quality meshes with a large number of faces while still maintaining a high framerate. In the DAVE environment the user can get near the mesh, making a high quality mesh even more important. Thanks to the recent upgrade the DAVE's computers have the latest hardware, thus supporting a high quality parameter.
The parameter has to be identical on each computer in the DAVE environment to make sure that all instances are running synchronously and use the same mesh. The mesh operations reference vertices by indices. Therefore vertex number ten, for example, has to be at the same position on all computers.
In theory, the quality parameter can be changed on-the-fly. However, each time the parameter is decreased, the mesh gets simplified. Thereby features like little bumps are destroyed and cannot be restored again.

3.5.5 Dynamic Tessellation

Based on the previously defined quality parameter, the preferred edge length is determined. By calling the edge split, edge collapse and edge flip operations, this edge length can be enforced globally. If an edge is too long it can be split, whereas short edges can be collapsed as they are not essential. Using the preferred edge length, the area of an ideal triangle can be calculated. Just because the edges are of the right length, it does not necessarily mean that the triangle takes up the right amount of space. If one angle is very large compared to the other two angles, the triangle will be rather flat and therefore have a comparatively small area. Removing it will result in just a small change in the mesh's appearance.

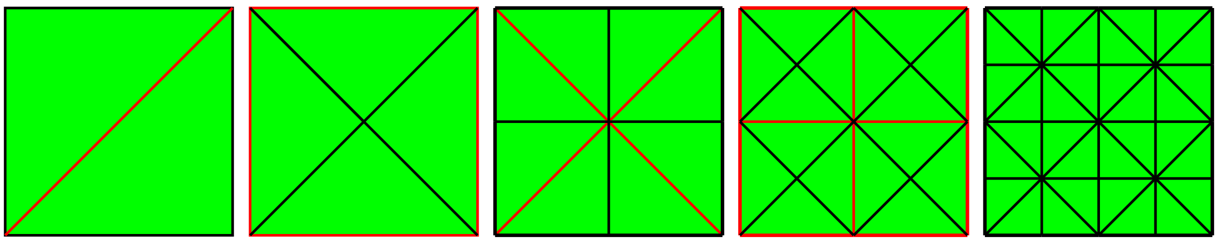


Figure 3.8: Calling edge split multiple times until all edges have the right length.

In order to maintain the right edge length throughout the whole mesh, the following procedure was devised:

- Keep track of all the parts of a mesh that were modified
- Perform checks for each modified face
  - If area is too large, call edge split using longest edge
  - If area is too small, or an edge is too short, call edge collapse

  – If an edge is too long, call edge split

- Back to the beginning, repeat if necessary

All checks use the preferred edge length as a reference. The area of an *optimal* triangle is based on this length. The order in which these checks are performed can be changed, resulting in slightly different tessellations. The procedure mentioned here proved to work reasonably well. In case one of the checks executes an operation, the resulting changes have to be stored and examined in a separate step. This leads to multiple calls of the same procedure, only with different sets of modified faces. The checks have to use thresholds to avoid infinite loops. If an edge is split, the resulting halves should be long enough to avoid being collapsed immediately afterwards. Otherwise the procedure will keep splitting and collapsing the same part over and over again. To make sure that this case can never happen, a safety precaution was implemented, limiting the number of loop iterations.

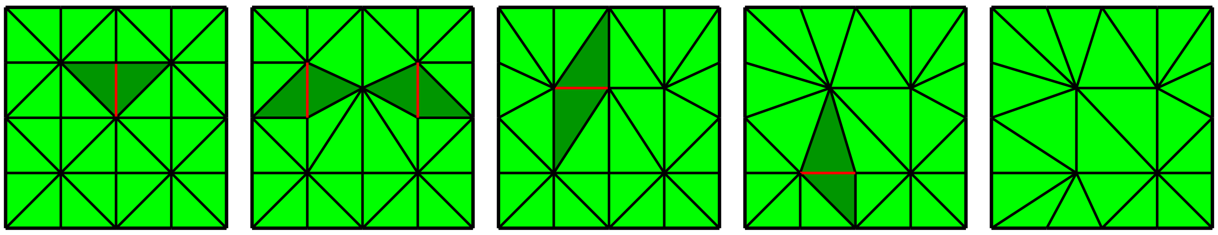


Figure 3.9: The edge collapse operation with a bigger preferred edge length will try to simplify the mesh again, while leaving the border edges intact. 25 vertices and 32 triangles at the beginning versus 20 vertices 22 faces in the end.

To further increase the mesh quality, the *centerVertex* operation can be called after each edge split or edge collapse. It will place the vertex of interest at a position that lies on the average of all adjacent vertices. As an average is calculated, the vertex can move in every direction. This creates a more regular tessellation and can reduce the amount of edge split and edge collapse calls, hence improving the performance. Using the centerVertex operation on a vertex that lies either on the border or on a sharp edge will lead to noticable changes and should therefore be avoided.

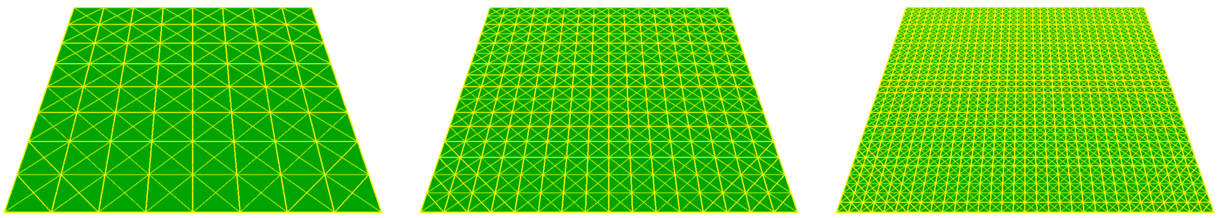


Figure 3.10: Starting off with nothing else but a single quad, the dynamic tessellation creates more or less uniform triangles. Different values are set for the quality parameter (0.5, 1.0 and 1.5). The higher the value gets, the smaller the triangles become.

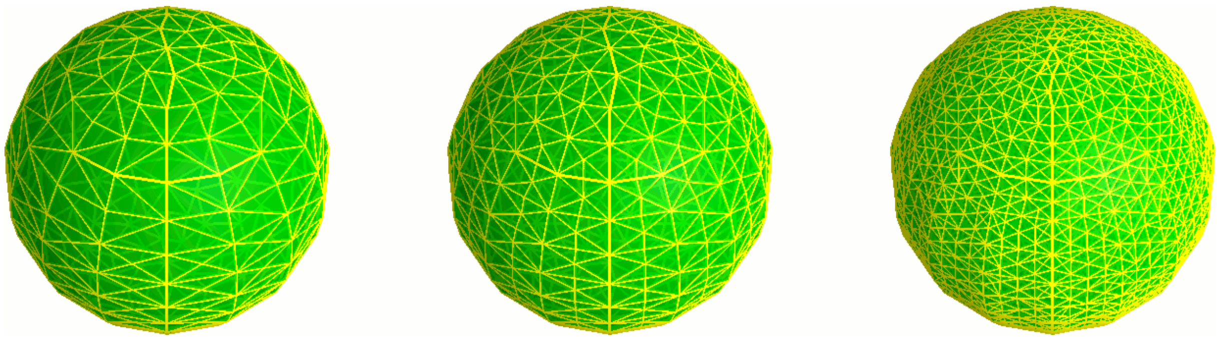
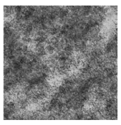


Figure 3.11: The same quality parameter values can also be applied to a sphere. However, a problem can be noticed when looking at the sphere's border. Even though the number of triangles in the third picture is very high compared to the previous pictures, the border still appears all but smooth. The *centerVertex* should be improved in order to solve this problem. Alternatively, the mesh could serve as a control mesh for subdivision surfaces. These surfaces could be created by utilising the Catmull-Clark algorithm [Cat74].

3.5.6 Multiple Textures

A single texture combined with vertex colours can provide a reasonable visual quality. In many cases this approach is sufficient. Depending on the object that is represented by the mesh, certain parts however can require a different kind of material in order to look right. Terrains for example do not necessarily have to be grassy everywhere. High mountains usually are covered with rocks, dry areas with sand. By changing the colour of grass, these materials can be approximated. If the user however decides to take a closer look, the rocks will not look like real rocks, and the sand only appears to be yellow grass. Blending multiple textures can therefore result in a better appearance. With each material using its own texture, the degree of realism can be improved. The tricky part is to add this feature to the existing mesh data structure without complicating it. By utilising the vertex colours, no additional information has to be stored. A colour consists of four different channels: red, green, blue and alpha. By mapping each channel to a specific texture, up to four textures can be used. A channel's value can be between zero and one. Therefore a colour with the values (1, 0, 0, 0) only activates the first texture, whereas (0.5, 0.5, 0, 0) combines the first two textures. The higher a channel's value becomes, the higher the influence of the corresponding texture gets. Using this technique, the vertex colours can either colourise the mesh the old-fashioned way, or provide the information that is necessary to apply multiple textures. The modelling tool has to decide which mode is activated.
The *texture blending* has already been implemented with the OpenGL shading language (GLSL). A small fragment shader programme can access both the textures and the vertex colours and therefore interpolate between the different values.

Texture	Color
	sand 1 0 0 0
	grass 0 1 0 0
	**rocks** 0 0 1 0

```
uniform sampler2D texture0;
uniform sampler2D texture1;
uniform sampler2D texture2;
varying vec2 tex_coord;

void main()
{
  vec4 tex0 = texture2D(texture0, tex_coord);
  vec4 tex1 = texture2D(texture1, tex_coord);
  vec4 tex2 = texture2D(texture2, tex_coord);

  tex0 *= gl_Color.r;
  tex1 = mix(tex0, tex1, gl_Color.g);
  vec4 col_tex = vec4(mix(tex1, tex2, gl_Color.b).rgb, 1.0);
  vec4 col_light = ambient + diffuse + specular;
  gl_FragColor = col_tex * col_light;
}
```

Figure 3.12: The table shows sample textures with the corresponding colour mapping. The accompanying GLSL fragment shader programme is listed on the right. Sand, grass and rock textures can be used to enhance the appearance of a terrain.

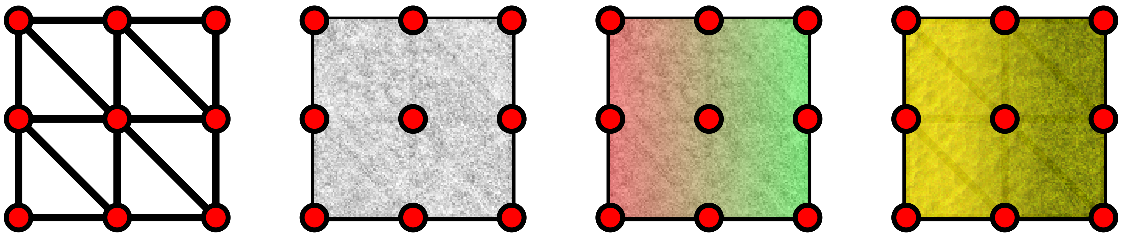


Figure 3.13: Small example that demonstrates the different appearances a mesh can have. The first picture shows all the edges and vertices, a mode that is mostly used for debugging purposes. The next two pictures show the same mesh covered by a single texture. First without and then with vertex colours. The last picture transforms these vertex colours into different textures using the multi texture mode.

3.5.7 Nearest Vertex

Finding the *nearest vertex* is an operation that has to be performed each time the selection is about to change. Hence it can be executed once per frame, and therefore many times per second. The easiest way to determine the nearest vertex would be to calculate the distance to each vertex. Of course, the nearest vertex has the shortest distance. This very naive approach depends on the number of vertices. Each added vertex increases the complexity. Checking all vertices can become very time consuming as soon as the mesh consists of more than just a few vertices. Modern computers however are fast enough to perform countless computations per second. The meshes this application usually has to deal with rarely consist of more than just a few thousand vertices. Trying to speed up the nearest vertex search can therefore be seen as an optimisation.

Instead of having to check all vertices in order to find the nearest one, it would be better to perform the search on a smaller subset. The easiest way to define unique subsets is to divide the space into separate areas. As a result, a vertex can only be part of one area at a time. This technique was implemented by using a regular grid that coincides with the floor. The grid divides the floor into

squares. Each square is part of a cuboid that reaches as far as the ceiling. As the vertices' distribution does not have to be even, some cuboids will contain more vertices than others. If a mesh is high rather than wide, most of the vertices will be part of the same cuboid. Hence the cuboids have to be split up into smaller parts. As a result, the whole scene is divided into regular cubes. The mesh data structure tries to enforce a certain triangle size. The number of triangles per cube should therefore not be too high. Hence, the size of one of these cubes has to be chosen in relation to the triangle size. A smaller triangle size will increase the number of cubes.
By using these subdivisions, it is possible to find the nearest vertex. If, however, there is no vertex in the selected subdivision, the search has to be expanded to neighbouring areas. It is sufficient to examine only the immediate vicinity. If there is no vertex within a certain range, the search can be aborted. One way of speeding up this process would be to use a so called *octree*. An octree evenly divides a scene, represented by a cube, into 8 sub-cubes. Each sub-cube can be subdivided again and so forth. The smallest subdivision corresponds to the previously defined cubes. These cubes store the actual vertices. The octree hierarchy only provides a means to quickly check large areas. In case an area contains vertices, a more thorough examination can be performed. As a result, the area of interest can be found faster. Not as many cubes have to be checked. This application currently does not use an octree as the search algorithm is fast enough for the time being. In case the quality parameter is set to a higher value and the number of vertices increases, the octree can still be implemented.
The actual nearest vertex search depends not only on a single position, but also on the viewing direction. These two parameters are provided by the optical tracking system. Starting from the joystick's position, a ray using the viewing direction is cast. By following this ray, the areas of interest are determined. In the next step these areas are combined in order to get a list of vertices. Each of these vertices can potentially be the sought vertex. The final step rates each vertex by combining the distance to the joystick with the distance to the ray. This technique acts like the spotlight selection used in the Chapel Hill Immersive Modeling Program created by Mark Mine [Min96]. Vertices that are near the joystick have to be very close to the ray, whereas vertices that are further away mostly depend on the distance to the joystick. Hence it becomes possible to select vertices that are further away without having to point at them very precisely.

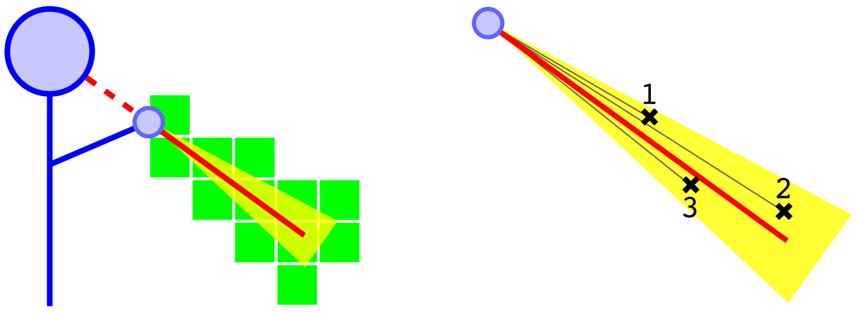


Figure 3.14: The diagram on the left depicts the area of interest selection. The green subdivisions are combined and examined in the second diagram. Three vertices remain. Vertex 1 has the smallest distance to the joystick, but is rather far away from the ray. Vertex 3 however lies almost exactly on the ray. Depending on the preferences, one of them will be picked. By default the search algorithm will prefer vertex number 1, as it is the nearest one. If vertex 3 is to be selected, the user will have to move more closely to it.

3.6 Synchronisation

3.6.1 Client / Server Architecture

In the real DAVE environment there are four stereo projectors, each displaying two images. One for the left eye, and one for the right eye. This results in eight different images that have to be rendered. As the rendering process is very time consuming, the work is distributed. There are eight instances of the same programme running on separate computers. They have to render the same scene, just from a different point of view. Therefore all the instances have to be synchronised in order to make sure that the scenes are absolutely identical. In addition to these eight instances, a special master instance is running on yet another computer, summing up to a total of nine computers. What makes this master instance special is the fact that it is the only one that has access to the tracking server and the input devices. Therefore this instance decides which operations are performed.

3.6.2 Network Protocol

3.6.2.1 UDP versus TCP

The server can send data to the clients via *User Datagram Protocol* (UDP) and *Transmission Control Protocol* (TCP). Each protocol has its advantages and disadvantages. With TCP, packets are guaranteed to arrive at the destination in the right order. To achieve this reliability, TCP sometimes has to retransmit lost packets, causing a delay. UDP on the other hand does not have to be reliable at all. It is especially well suited for time-sensitive applications that prefer to drop packets when they are not arriving in time. By doing that, unnecessary delays are avoided. When new and updated data is continuously transmitted, the loss of a single packet is negligible. Common applications are voice over IP or any other kind of media streaming. If the packets do not arrive in time, they can be skipped.

3.6.2.2 UDP SyncBuffer

Based on the data received from the tracking server, the user's position and rotation are set. By adding an offset, the left and right eyes' positions can be calculated. The user will not be standing still, therefore a constant update is necessary. Usually the scene is rendered at a rate of sixty images per second. An update might be valid for only one of these images. Retransmitting an update in case of an error seems pointless, as this update is immediately outdated. This makes UDP the perfect choice, even though it is comparatively unlikely that errors occur in the DAVE's local network. By utilising UDP's multicast capability, the data only has to be transmitted once.
An update consists of the so called *SyncBuffer* structure and optional user data. It can be identified by a *magic number* which occupies the first bytes. The included eye positions correspond to the position of the shutter glasses and are required for the rendering process.

```
struct SyncBufferStruct
{
  char magicNumber[4];
  float eyePosLeft[3];
  float eyePosRight[3];
}
```

As the modelling application has to transmit extra information such as the position of the joystick, additional data is stored in the *SyncBufferUserData* structure. This structure is appended to the SyncBuffer. Together, they comprise all data necessary to update the rendering of the scene. The difference between *joystickPos* and *joystickDisplayPos* is that joystickPos corresponds to the actual position of the joystick whereas joystickDisplayPos corresponds to the position of the cursor. These two positions do not have to coincide.

```
struct SyncBufferUserData
{
  float rotation[4];
  float joystickPos[3];
  float joystickRotation[4];
  float joystickDisplayPos[3];
  float joystickDisplayRotation[4];
}
```

Depending on the computer's settings, several hundreds of images can be rendered per second. The differences between two succeeding images will be marginal. It will suffice to send an update only every ten or twenty milliseconds. The benefit of such a fixed update interval is that the amount of traffic will be reduced considerably and it is less likely that the network will get congested.

3.6.2.3 TCP Packages

Besides the constant SyncBuffer updates via UDP, there are TCP connections between the clients and the server. The packets transmitted from the server contain commands that have great influence on both the mesh and the environment. Instead of transmitting the entire mesh each time the user performs an operation, only the commands have to be sent. Arrival of the commands must be guaranteed and their order is of great importance. Swapping two separate mesh manipulations would result in a different mesh. TCP complies with all of these requirements which makes it the perfect choice for transmitting commands. In contrast to the SyncBuffer that always has an identical structure, commands can have varying lengths and contents. Hence each command requires special handling. First of all commands have to be identified before being further processed. A protocol has been devised, to divide the commands into separate categories. The main categories are: system, modeller, tool selector and info screen. Each category has its own packet handler which can detect and process incoming packets.

Header	SubHeader	Parameters

Table 3.1: TCP command structure: header containing the category, followed by the subheader which contains the command identifier and finally the command's parameters.

Commands can be sent one after the other. Separators are added by the recipient between data sent from different computers. Such separators contain the sender's ID and the amount of bytes that were transmitted. This enables sender dependent handling. In case of an error, only data to the next separator is lost.

Separator	Command	Command	...	Separator	Command	Command	...

Table 3.2: Incoming TCP data structure: Separators followed by multiple commands

The categories and their commands are:

- **SEPARATOR**
- **SYSTEM**
  - SERVER_SHUTDOWN: force shutdown
  - TOGGLE_DEBUG_MESH: toggle mesh debug visualisation
- **MODELLER**

  - CREATE: create new modeller
  - ACTION: perform modelling operation
  - SET_EDIT_MODE: set current edit mode
  - SET_SELECTED_PIECE: change selection
  - SET_DRAGGING: toggle dragging mode
  - SET_DRAGGING_PARAMS: set dragging parameters, e.g. direction
  - SET_IN_USE: (de-)activate modelling operations
  - SET_COLOR: define main colour

- **TOOL_SELECTOR**
  - MODE: change set of tools
  - SELECT: select a certain tool by enlarging and rotating it
  - HIGHLIGHT: special highlighting of active tool (e.g. current colour)

- **INFO_SCREEN**
  - MESSAGE: display a message
  - TEXTURE: display a texture
  - FIXED: change *fixed* attribute

The amount of network traffic is reduced by compressing the data using the open source *zlib* [DG96] library. Depending on the contents, the data can be usually compressed by up to fifty percent, justifying the extra computations necessary for the compression. However, the amount of network traffic is very low. Even if the user is continuously modifying the mesh, only a few kilobytes are transmitted per second. Therefore compression is not absolutely necessary.
Protocol and package handlers were designed for easy extension. Additional functionality can be added without interfering with the already existing structure. Different protocol versions are compatible to some degree. As certain packages can change and require different handling, only identical applications should be used in the first place. Therefore the server will only accept connections from clients which are the same version. This can be ensured by the special handshake process. Client and server exchange some messages containing data such as the identification and version of the application. The server can then decide whether to accept the client or not.
As of now all packages are sent from the server. The clients only have to listen to what the server has to tell them. Maybe at some point the clients will have to send packages as well. Hence the server has to process incoming packages, just like the clients. This makes the server vulnerable to attacks. If a modified client sends the shutdown command and the server complies, one might be tempted to exploit it and cause trouble. For this reason the protocol can mark packages as server only. If a client sends such a package, it can safely be assumed that this is not normal behaviour. The connection can therefore be terminated and the threat eliminated.

Chapter 4

Results and Discussion

4.1 Results

4.1.1 Pipe Modelling

The pipe modelling tool was the first tool created for the modelling application. It enables the user to move an input device in 3D space and create sketches. Only basic functionality was implemented as blob-based tools became more important. It can therefore be rather complicated to create extensive sketches.

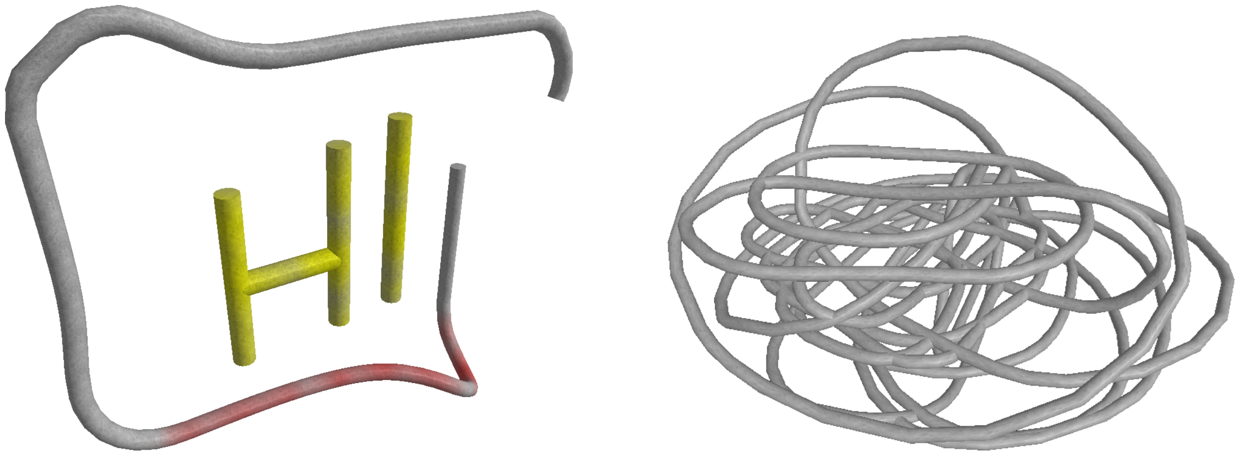


Figure 4.1: Sketches can be created by combining multiple pipes (left hand picture). However, space is rather limited. Too many pipes tend to form giant knots, as demonstrated by the right hand picture.

In order to improve this particular modelling tool, the radius of the pipes must be decreased and consequently a single pipe will require less space. The number of pipes can therefore be increased. Handling these pipes will be difficult without implementing the advanced features mentioned in the design chapter. Being able to modify and interact with existing pipes will be mandatory if complex scenes are to be created. Presently, existing pipes can only be extended. If a pipe does not have the right shape, the user only has the possibility to delete everything and start from scratch.

4.1.2 Landscape Modelling

The landscape modelling tool turned out to be the most interesting tool of all. Despite its limited functionality, users can generate complete landscapes full of hills in a matter of seconds. However, depending on the mesh quality parameter and the resulting number of triangles, edges can become visible. In order to avoid this problem, the cursor always selects multiple vertices at the same time by defining a minimum radius. Each transformation can therefore utilise these vertices to create a smooth surface.

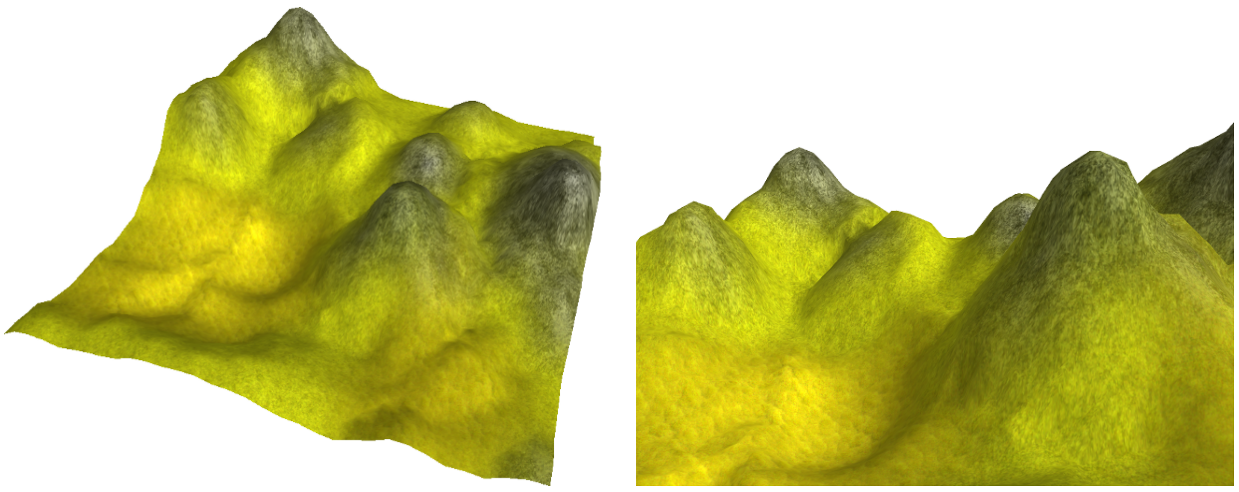


Figure 4.2: A landscape created by the modelling application. The left picture was taken from a great height. From this point of view the user can perform edit operations all over the landscape. The DAVE also enables a user to take a closer look as shown in the right picture.

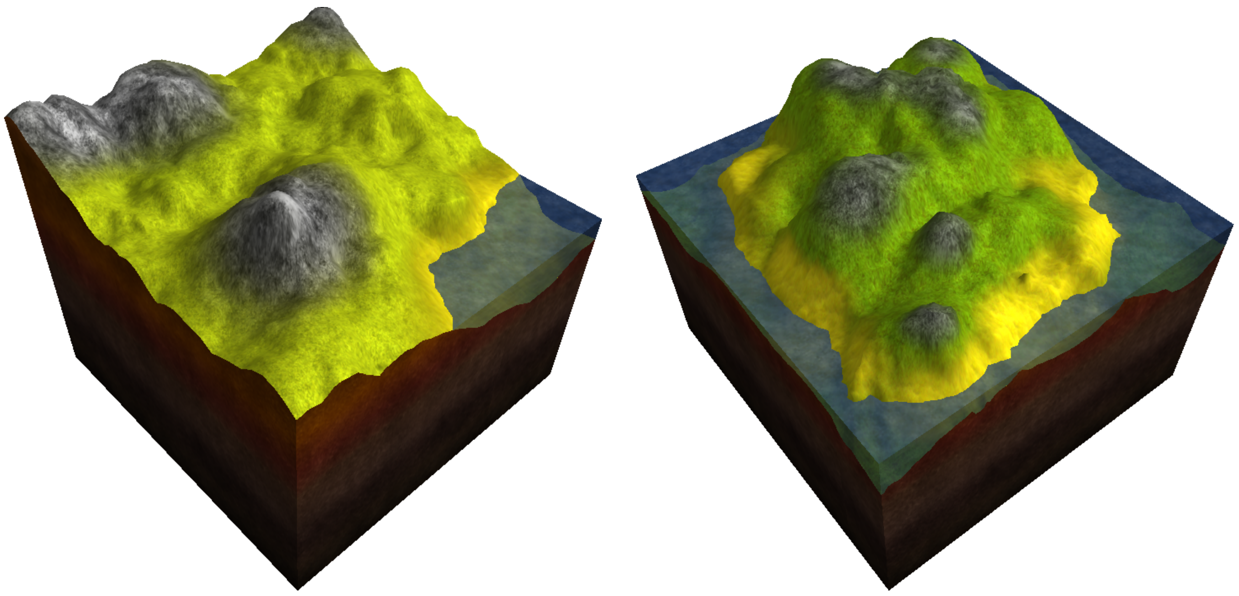


Figure 4.3: More pictures demonstrating the capabilities of the modelling tool. A landscape can contain a lot of small hills or consist of one large hill. By adding a layer of water, the hill in the picture on the right is turned into an island.

4.1.3 Blob Modelling

Although the landscape modelling tool is similar to the blob modelling tool, the resulting sculptures and shapes are very different. The landscape tool limits edit operations to a certain direction whereas the blob tool allows arbitrary manipulations. This makes it particularly difficult to find proper texture coordinates. Blob-based sculptures currently have no texture at all. Textures can make objects look more detailed than they actually are. As a result, the appearance of blob-based objects greatly depends on the quality of the mesh. Edges are likely to be more visible.

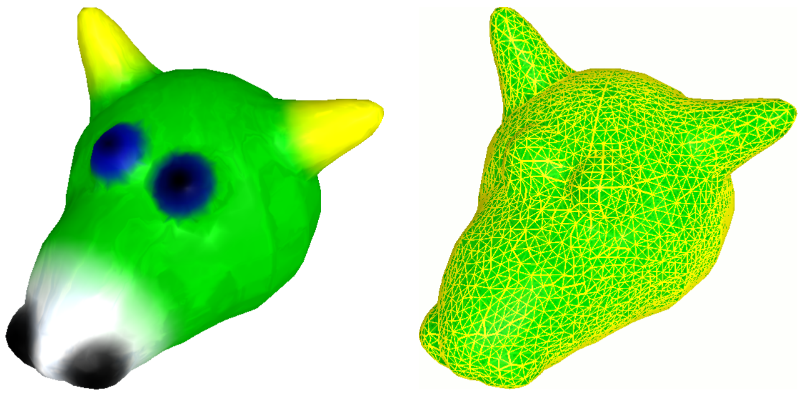


Figure 4.4: Just a few operations were required to transform an ordinary sphere into this animal-like sculpture. By adding colours different parts of the head, for instance the eyes, become more obvious.

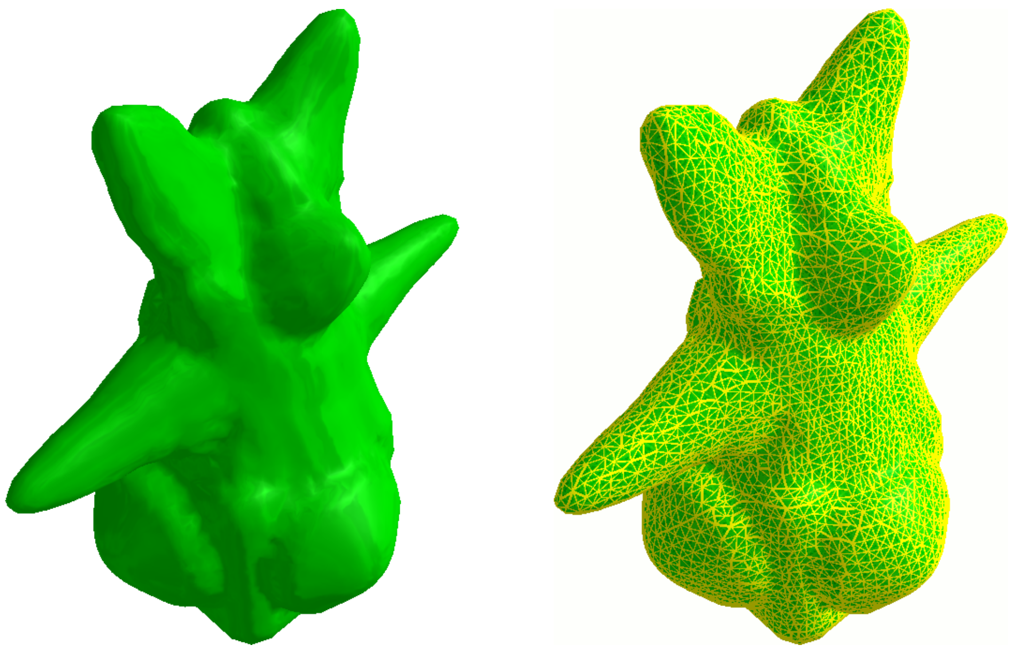


Figure 4.5: The flexible blob modelling tool enables users to create arbitrary organic shapes. The picture on the right shows the fairly regular triangulation.

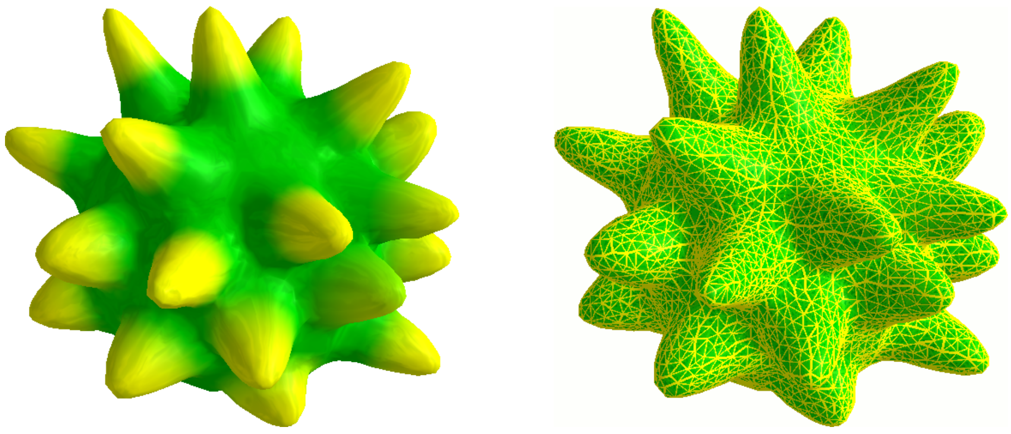


Figure 4.6: Another blob-based mesh. Each peak was created by executing a single operation. As a result, the entire mesh was created in just a few seconds.

4.1.4 Modelling Tool in Action

It is difficult to demonstrate the capabilities of the modelling application. Normal screenshots are inadequate because they only show static contents of the screen. To highlight the specialities of this application it is necessary to show the user as well. The following photographs, taken by Marcel Lancelle, show the modelling application running inside the DAVE. Everything appears to be double due to stereoscopy. The scene is rendered from the point of view of the camera rather than of the user which leads to less noticable distortions.

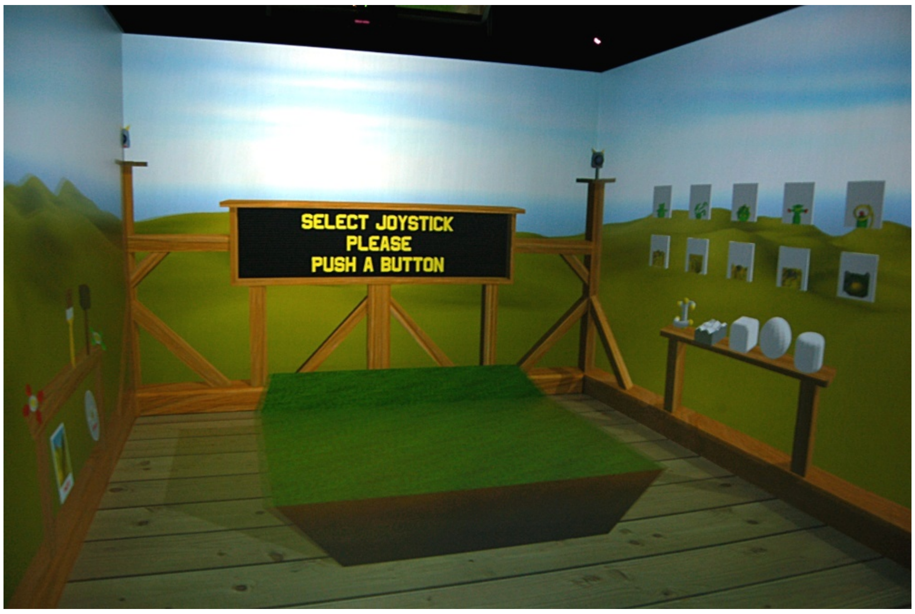


Figure 4.7: This photographs shows the modelling application right after it has been started. The info screen instructs the user to select a joystick by pressing a button. This step is necessary as multiple joysticks can be connected to the master computer simultaneously. As soon as the user complies, objects can be created and manipulated. By default, the landscape modelling tool is activated, recognisable by the flat green surface.

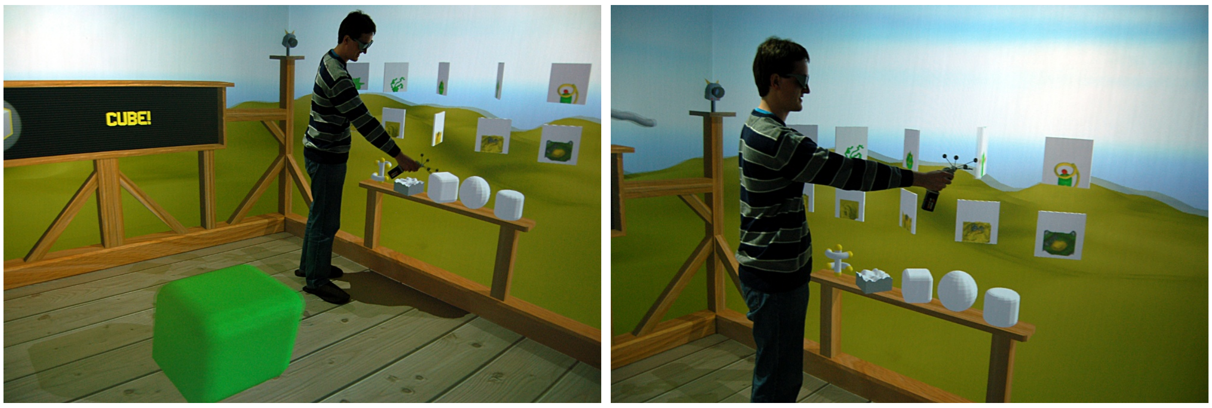


Figure 4.8: The user can either select one of the predefined shapes, located on the shelf in the left picture, or load a previously created object, represented by the flat white cuboids in the right picture. These previously created objects are symbolised by their images. The modelling application automatically stores these images alongside the meshes.

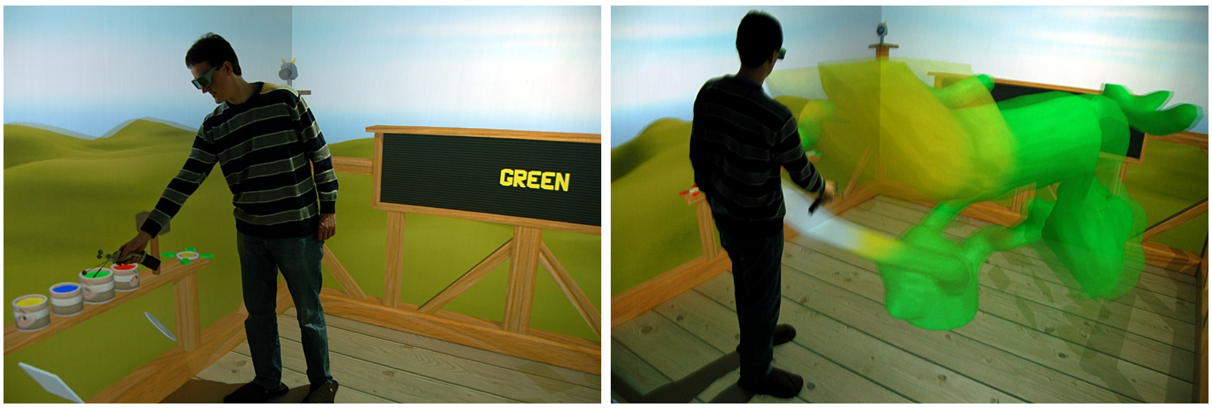


Figure 4.9: As soon as a shape has been selected, the user can walk to the opposite side and chose a tool. The picture on the left shows the selection of the colour green which will enable the user to colourise the object. By pointing at a specific part of the object and pressing a button of the joystick, the colour will be applied, as shown in the right picture.

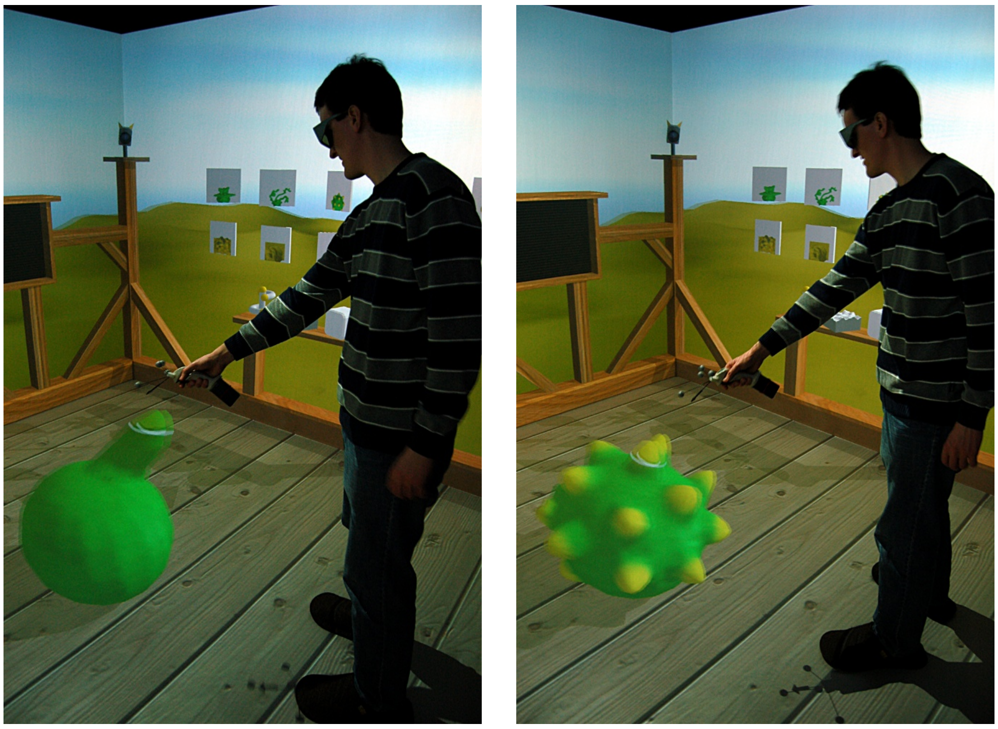


Figure 4.10: The sculpt tool changes the shape of an object. It enables the user to grab a point and pull it in any direction, shown in the picture on the left. Arbitrary objects can be created by combining this sculpt tool with the colour tool. Just a few operations were necessary to transform a regular sphere to the object in the picture on the right.

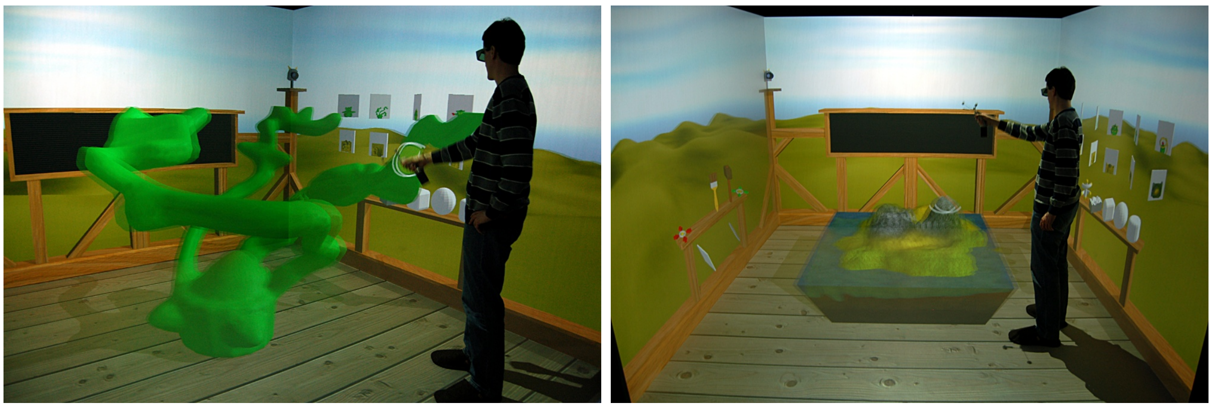


Figure 4.11: By altering the size of the cursor more complex objects can be created, as shown in the picture on the left. This feature is also important for the landscape modelling tool. The picture on the right shows an island surrounded by water. The two large hills were created by setting the cursor size to a high value.

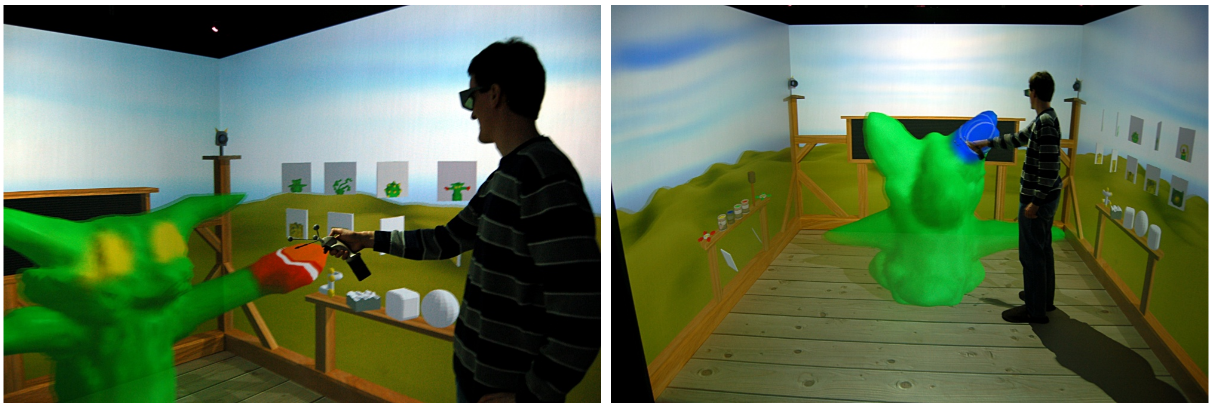


Figure 4.12: More objects demonstrating the capabilities of the modelling application. It can be difficult to perform precise operations. As a consequence, two objects never look alike.

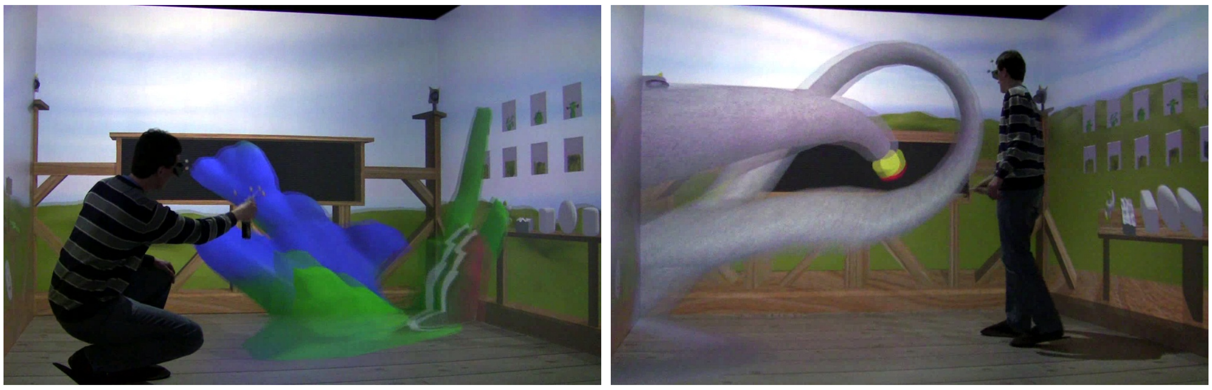


Figure 4.13: In these two pictures the scene was rendered from the viewpoint of the user. Everything appears to be distorted when looking from a different point of view. As a consequence, it is very difficult to recognise the actual shape of the object from outside the DAVE.

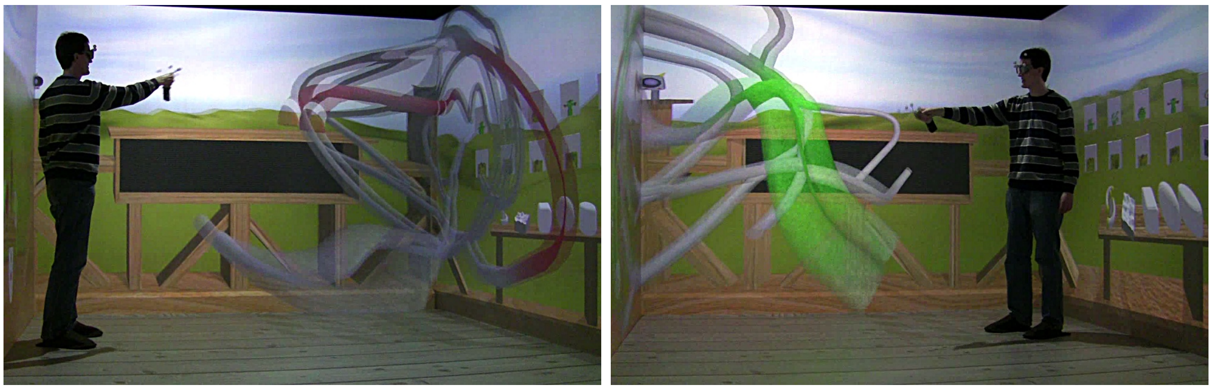


Figure 4.14: Although the pipes in picture on the left appear to take up almost the entire space, there is still enough space left as soon as the user walks to the opposite side, as demonstrated by the picture on the right. The green and red pipes are very near to the user as the pictures for the left and right eyes vary significantly.

4.2 Problems

4.2.1 Mesh Representation: from 2D to 3D

At the beginning, the mesh was limited to a single plane. Testing mesh operations on a flat surface seemed to be a reasonable choice. No camera had to be adjusted in order to see the important parts. The mouse could be used as the primary input device. It could easily be determined if the mouse cursor was currently inside the mesh or not. In case it was outside and the distance was within a certain threshold, a triangle was added. Hence the mesh could be enlarged and take up any shape. If the threshold had the right value, the triangulation automatically became fairly regular. By merging near vertices, even gaps could be closed. Early tests behaved similar to a regular drawing programme. Instead of placing pixels on a canvas, a connected mesh was created.

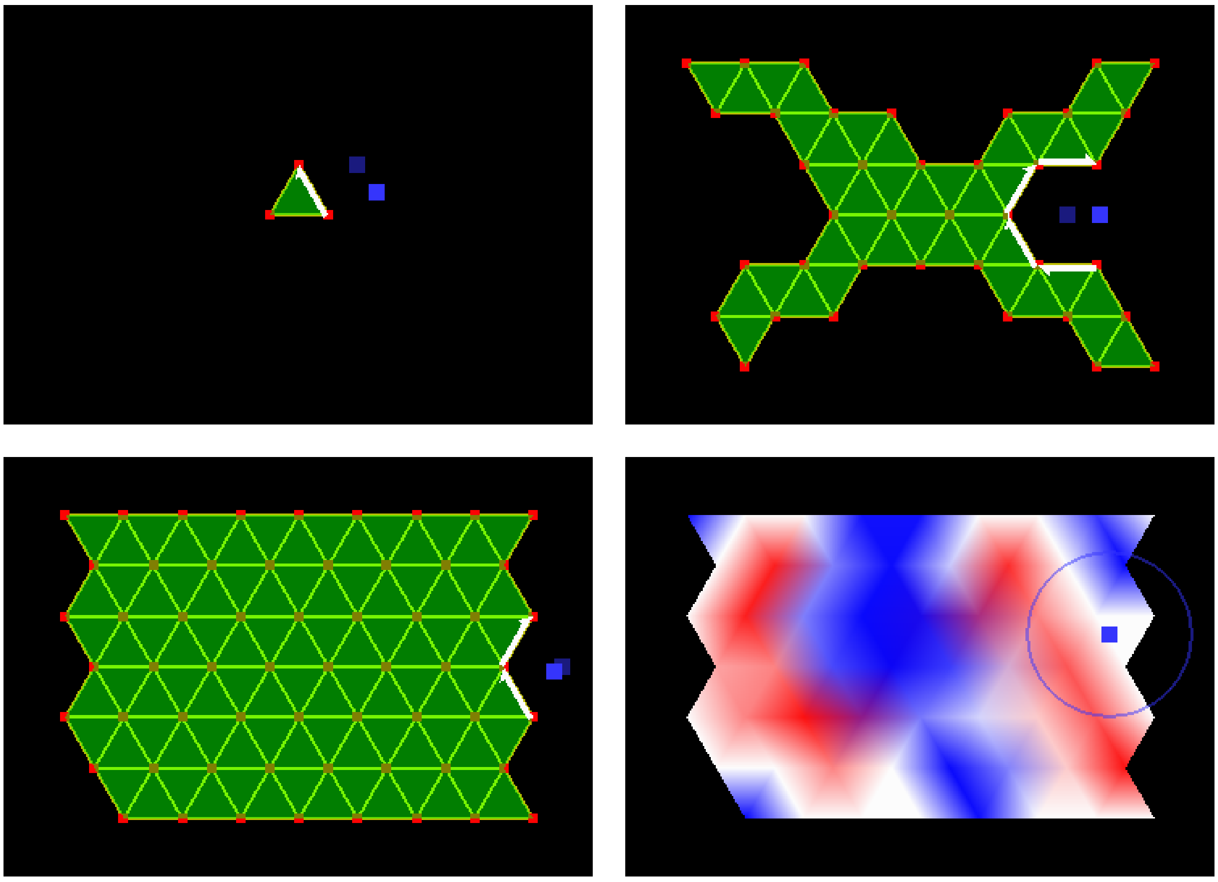


Figure 4.15: A series of screenshots showing the 2D mesh data structure test application. Starting off with a single triangle, the mesh can be extended in any direction. The light blue square depicts the mouse cursor's position. The dark blue square is the suggested position for a new vertex. This position is calculated by adding an offset to the adjacent edges. These edges are marked by white arrows. A new triangle will be connected to each one of them. An alternative editing mode allowed the mesh to be coloured by assigning a colour to each vertex. The triangulation is very regular due to the fact that the starting triangle is equilateral. Arbitrary triangles lead to a less uniform triangulation.

However, transforming these operations into 3D space proved to be rather difficult. Adding a third dimension not only added extra complexity, it also created new problems. Operations that worked perfectly well in 2D space now generated awkward looking shapes. The problem was that new triangles were added to border edges. In 2D space, these new triangles automatically lie on the *right* plane.

However, there is no such thing as a *right* plane in 3D space. The triangle only has to be connected to an existing edge. Everything else is arbitrary. Hence, triangles started to intersect. Many parts of the mesh overlapped. The surface was not closed and contained many holes. Each added triangle seemed to make it worse. The whole way of adding triangles to border edges seemed to be flawed. It was impossible to create proper meshes. Creating separate strokes consisting of triangle strips seemed to be the only reasonable application. Hence it was time for a fundamentally different approach.
Instead of adding each triangle separately, a complete mesh should automatically be constructed. Depending on the input of the user, the mesh was modified. Therefore the user does not have to worry about details like closing holes in the surface. The user can concentrate on shaping the object. As the shapes, however, can vary greatly, special tools had to be devised. Each tool had its special purpose. One tool was responsible for creating pipe-like objects, another one for creating landscapes. The third and final tool, called blob, was more generic and could create virtually every shape. It was completely different in that it did not rely on a special mesh structure. While the pipe modeller manually extruded fixed circles, this new tool could modify any shape, thus making it superior and more flexible. Modifications to a mesh were performed by moving parts. If triangles became too large, they were subdivided. Small triangles were removed completely by enlarging the adjacent triangles. Hence, a more or less uniform triangulation could be enforced.
By adding the ability to define sharp and smooth edges, more advanced operations could be implemented. As the development progressed, the landscape tool was eventually replaced by a modified version of the blob tool, resolving some of the inherent weaknesses. The pipe modelling tool remains the only tool that works in a completely different way. A blob tool could potentially be used to create objects that resemble pipes. The resulting meshes however would not be as smooth and round. Little dents and bumps here and there would be inevitable. Hence these pipes would appear to be organic rather than man-made.
This bumpiness leads to another problem all blob-based meshes have to deal with. Special operations try to enforce a mesh consisting of more or less uniform triangles. However, depending on the surface, this can lead to unwanted artifacts. If the surface is very bumpy, the triangle size might be too large in order to properly represent all of the surface's features. One way of solving this problem would be to allow different triangle sizes. Flat surfaces require a smaller number of triangles. Smooth transitions and curves however would benefit from a reduced size. The necessary information could be extracted by calculating the angles between the faces. This operation has already been performed in order to detect sharp edges. If the resulting angle is very small, the surface is rather flat. Hence the triangle size can be increased. However, a nonuniform triangulation will be noticable when the user is trying to colourise the mesh. Areas consisting of smaller triangles naturally have a larger number of vertices and more detailed drawings can be placed on the surface. Thus, it will be impossible to create the same drawings on a flat surface. The comparatively small number of vertices that comprise a flat surface simply cannot store the required information. As long as colours are applied on a per-vertex basis, a uniform triangulation is necessary.

4.2.2 Texturing

Creating texture coordinates for arbitrary objects remains an unsolved issue. Only the pipe modelling tool as well as the landscape modelling tool can calculate valid texture coordinates. However, their functionality is rather limited. The resulting objects always share a common structure. Pipes consist of cylindrical segments. Adding a segment just requires the tool to extend the last texture coordinates. Landscapes are deformed quads, the texture coordinates do not have to be modified at all. The blob modelling tool however can create almost any shape. Hence the texture coordinates cannot be calculated as easily. Currently the blob tool just deforms the existing coordinates. As long as the modifications are not too severe, the result is still tolerable. However, sooner or later the texture coordinates are unusable and the texture will look completely distorted. Therefore a blob usually does not have a texture and only displays the vertex colours.

Creating texture coordinates with a shader programme or the built-in OpenGL function *glTexGen* [Sil06] makes it difficult to store the coordinates as they are calculated on-the-fly. These coordinates have to be stored in some way if the application is supposed to be able to export a mesh. Transferring data from the graphics hardware back to the central processing unit is rather complicated.

4.2.3 Synchronisation

Modifications done to a mesh can be quite extensive. Even if only a small part of the mesh is moved, the resulting changes influence the whole surrounding area. Usually the user selects parts of the mesh and translates them to a different position. As soon as this operation is completed, the tessellation is optimised in order to enforce a uniform triangulation. Large triangles are split into two triangles, small triangles are deleted. Hence the structure can change quite noticably. However, all application instances have to perform these changes in exactly the same way. Even the smallest difference can lead to inconsistencies in time. If the master application instructs the clients to move vertex A to a new position, this vertex might not even exisist on all client instances. This will either cause the programme to crash, or at least make it useless as it no longer displays the correct mesh. In general, all applications are supposed to perform identically. The selection is calculated based on the cursor's position and its radius. Subsequent transformation is set by the joystick's movement. As these instructions can easily be transferred to all instances, they should not cause any problems. However, transformation is followed by mesh optimisation. This optimisation can cause infinite loops and therefore must be interrupted. Nevertheless, all applications should perform the optimisations in the same way.
An ill-considered optimisation in the network handling part caused this procedure to fail. Mesh optimisation was called as soon as the incoming network traffic was handled. Usually, this will work as expected. However, as soon as the mesh gets more complex and the framerate starts to drop, more than one mesh modification can end up being in the incoming network buffer. Hence the second modification will operate on non-optimised data and therefore result in the *wrong* mesh. From then on things just got worse. Sooner or later differences were too severe and the programme crashed.
Now that this bug has been repaired, the synchronisation works considerably better. However, there still seem to be some flaws left. Instead of relying on consistent data, each modification sent to the clients could contain extra information. This extra information could provide the means to ensure that all modifications are performed on identical data. In case differences are detected, modifications should not be performed. Currently, an action requires the following data:

```
struct Action
{
  int button;
  float position[3];
  float rotation[4];
  float radius;
  int vertex_id;
  int piece_id;
}
```

The joysticks state is specified by transmitting its position, rotation and the pressed button. The radius as well as the vertex and piece identifiers determine the selection. The piece identifier is only required by the pipe modelling tool and will be dropped as soon as the tool becomes blob-based. Redundant information such as the mesh's checksum or the selected vertex's position can help to perform checks. If this position does not coincide with the *real* position, the operation should be aborted. The mesh is therefore erroneous and should no longer be used. However, the client must be able to recover. As soon as the master application is informed, it will transmit the whole mesh. Thereby the damaged mesh can be replaced.

The proposed extended action information is:

```
struct Action
{
  int button;
  float position[3];
  float rotation[4];
  float radius;
  int vertex_id;
  float vertex_position[3];
  int vertex_degree;
}
```

The position of the vertex as well as its degree (the number of vertices it is connected to) should suffice to detect differences in the mesh.

4.2.4 Exporting Arbitrary Meshes

Exporting a mesh in a way that preserves all of its features proved to be rather complicated. Every file format has its advantages and disadvantages. The currently preferred PLY format lacks support for smoothing groups or any other means to specify if an edge is supposed to be sharp or smooth. This information is implicitly defined by normal vectors. If an edge is sharp, the two adjacent faces have different normal vectors. Nevertheless, these faces share two common vertices. As normal vectors are defined on a per-vertex basis, these two vertices have to be defined twice: once for each face. Hence the resulting file will contain some vertices that coincide. They can only be distinguished by their normal vectors. As a result, the two faces will no longer appear to be connected. Even though their vertices coincide, they are separated. In order to merge these vertices again, post processing steps have to be performed. The following steps are necessary when using Blender:

- Import PLY file
- Select all vertices and press *Remove Doubles*
- Activate smooth shading by pressing *Set Smooth*
- Add *EdgeSplit modifier* [Bleb]
- Adjust split angle parameter

The resulting mesh still looks like the original mesh. However, it is now seamlessly connected. The EdgeSplit modifier detects sharp edges based on the angles between faces. Completely smooth surfaces such as landscapes do not require the EdgeSplit modifier as no sharp edges exist. The mesh data structure has to repeat similar steps in order to re-import meshes:

- Load PLY file
- Merge coinciding vertices
- Calculate normal vectors and detect sharp edges

However, as sharp edges are detected on-the-fly in both cases, they cannot be defined in a file explicitly. Meshes created by the modelling programme usually consist of smooth rather than sharp edges. Therefore this problem does not have too much influence on the functionality.
This little example demonstrates the way smooth and sharp edges are stored. The mesh consists of two quads sharing a common edge.

```
ply
format ascii 1.0
element vertex 8
property float x y z nx ny nz
element face 2
property list uchar uint vertex_indices
end_header
1.000000 −0.000000 1.000000 0.000000 0.707107 0.707107
1.000000 1.000000 0.000000 0.000000 0.707107 0.707107
−1.000000 1.000000 0.000000 0.000000 0.707107 0.707107
−1.000000 0.000000 1.000000 0.000000 0.707107 0.707107
−1.000000 0.000000 1.000000 −0.000000 −0.707107 0.707107
−1.000000 −1.000000 0.000000 −0.000000 −0.707107 0.707107
1.000000 −1.000000 0.000000 −0.000000 −0.707107 0.707107
1.000000 −0.000000 1.000000 −0.000000 −0.707107 0.707107
4 0 1 2 3
4 4 5 6 7
```

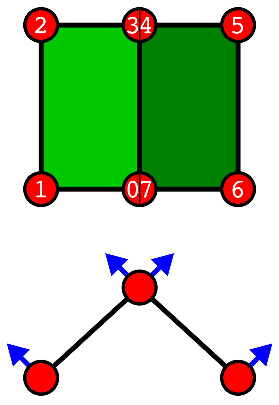


Figure 4.16: If all edges are defined as sharp, each face has its own set of vertices. Eight vertices are stored in order to represent two faces. However, vertices 3, 4 and 0, 7 coincide. The drawing on the bottom shows their normal vectors.

```
ply
format ascii 1.0
element vertex 6
property float x y z nx ny nz
element face 2
property list uchar uint vertex_indices
end_header
1.000000 −0.000000 1.000000 0.000000 0.000000 1.000000
1.000000 1.000000 0.000000 0.000000 0.707083 0.707083
−1.000000 1.000000 0.000000 0.000000 0.707083 0.707083
−1.000000 0.000000 1.000000 0.000000 0.000000 1.000000
−1.000000 −1.000000 0.000000 0.000000 −0.707083 0.707083
1.000000 −1.000000 0.000000 0.000000 −0.707083 0.707083
4 0 1 2 3
4 3 4 5 0
```

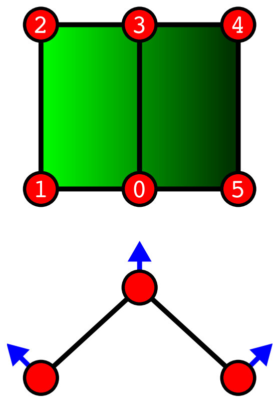


Figure 4.17: If the shared edge is smooth, the two faces can share vertices. Hence the total number of vertices is reduced by two. The normal vectors are merged.

4.2.5 Cursor Visualisation

The position of the cursor is calculated using the nearest vertex algorithm. Hence the position can only coincide with one specific vertex. As the cursor has a radius, adjacent vertices become part of the selection. This selection is visualised by drawing a circle. By moving the joystick, the circle will start to jump from vertex to vertex. Depending on the mesh quality and the resulting triangle size, the jumps can become quite noticable. If the triangles are too large, the circle's centre might be far away from where the user is actually pointing at. Due to the optical tracking system's lack of precision, the position of the joystick can vary. Hence the selection can change even though the user does not move at all. Most of the time, the user will not be pointing at a specific vertex. The ray used by the nearest vertex algorithm actually intersects the surface somewhere inside of a triangle. Hence by moving the joystick and therefore altering the direction of the ray, this point of intersection is going to move as well. Even if it stays inside of the same triangle, the selection can change. Any one of the three vertices forming this triangle can be selected. If the ray intersects the triangle near its centre of gravity, the selection becomes very unstable. The slightest move will cause the selection to change. Hence the cursor will appear to be very jittery.

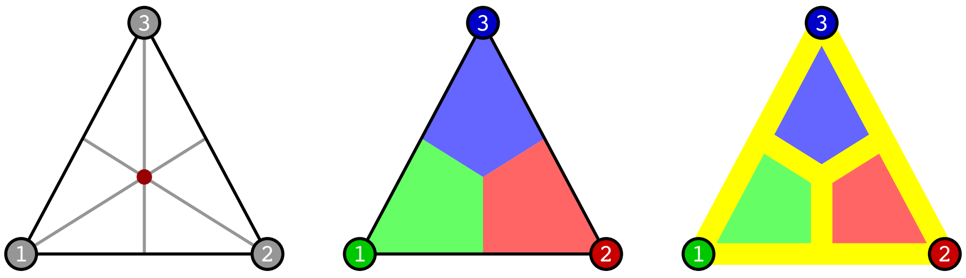


Figure 4.18: The centre of gravity can be calculated by intersecting at least two medians. These medians can also be used to determine which vertex gets picked. If the ray intersects the triangle inside of the green area, vertex number one will be selected. If, however, the ray-triangle intersection point lies anywhere near a median or one of the triangle's edges, the selection can jump to a different vertex. The yellow lines symbolise this *dangerous* area. It covers a great part of the triangle.

In an attempt to solve this issue, three methods were added to the mesh data structure. They require a ray in order to work, just like the nearest vertex algorithm. This ray is defined by the originating position and a directional vector.

- **Face::isIntersected**: true if ray intersects the triangle
- **Face::getIntersection**: return intersection of the triangle plane and the ray
- **Mesh::getInterpolatedPosition**: get position on the mesh

These methods are meant to improve the result provided by the nearest vertex algorithm. The *getInterpolatedPosition* method uses this nearest vertex to calculate a position on the surface of the mesh that coincides with the ray-surface intersection. Based on the nearest vertex, all adjacent faces are examined. If one of these faces is intersected by the ray, the intersection point is calculated using the *Face::getIntersection* method. Hence the cursor can move freely on the surface of the mesh. The size of the triangles no longer has an influence.
However, this approach does not work particularly well for an arbitrary mesh. Very often, none of the adjacent faces are intersected by the ray. Hence the interpolated position cannot be calculated and the cursor starts to jump again. A more sophisticated approach has to be devised. The current approach is only suited for a more or less flat surface. In addition, border edges are not handled. The

landscape modelling tool currently is the only tool that creates meshes with border edges. If the ray does not intersect the mesh, it is likely to be near a border edge. Instead of interpolating the position on a surface, it would be sufficient to interpolate along the border edge. This, however, requires a different kind of calculation.
Currently, the interpolation algorithm is deactivated. The plane nearest vertex algorithm is preferred as long as not all problems are dealt with. An alternative to the complicated calculations would be to directly interpolate between the results provided by the nearest vertex algorithm. If a vertex is selected, the cursor will slowly move to the new position. Hence the discontinuity is replaced by a smooth transition. However, the cursor will only move on a straight line between the old and the new position. If these positions are on opposite sides, the cursor will seem to penetrate the mesh. Therefore the smooth transitions should only be acivated for adjacent vertices. In addition, the transition should be executed in a very fast way. If the cursor seems to lag, the user will get the impression that the programme itself is slow.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

As mentioned in previous chapters, the DAVE can be used for many different applications. It provides a whole new way of interacting with a virtual environment. Instead of using a monitor as a window to look inside of this environment, one can actually walk around and be more independent. Utilising these new possibilities is a major challenge. There are countless applications available that can be used to create meshes. However, almost all of them are not designed for this kind of environment.
It took some time to get used to this way of interacting with an application. As a result, the first tests were barely usable. They served as a means to implement basic functionality such as synchronising multiple instances and accessing input data. As time progressed, more and more features were added. Instead of a single triangle floating in space, connected meshes could be created. The final step was to turn the surroundings into something other than plain black. Now the user could walk around in an immersive environment and create arbitrary objects.
Ongoing tests were performed in order to reveal flaws in the programme. Users who had no experience in using this application were able to point out shortcomings in the user interface. For example, the tool selection caused a few problems. This selection is based on a simple calculation. The nearest tool is determined by the position of the joystick and the position of the glasses. However, users tended to select a different tool by accident. Just by standing either on the left or right side, the user was close enough in order to select a tool. For example, instead of performing a modelling operation by clicking a button on the joystick, an entirely different model was loaded. This, of course, was never intended. Therefore a small but important change in the programme was implemented. Currently, the user has to look and point at a tool at the same time in order to select it. Problems such as this can happen very easily as an application for the DAVE has unique requirements. Even though countless tests were performed in the simulator, the problem remained unnoticed. This shows how important *real* tests are.
New users only require very brief instructions. Switching between different types of modelling tools becomes obvious after a few clicks. Landscapes and sculptures can then be created in a very short amount of time. As as result the modelling application can serve as an interesting way of demonstrating the capabilities of the DAVE and enable a user to immerse into an interactive 3D scene.

5.2 Further Development

It is difficult to tell if an application is finished. There are certain aspects which can be changed and insert features that are missing. Even though many of these extra features are not essential, they can improve the overall quality of the software and thereby make it more usable.

5.2.1 Use Modelling Tools Simultaneously

The application can only use one modelling tool at a time. A user can either create a landscape or draw sketches. Switching between these tools will erase the existing scene. It is therefore impossible to extend a landscape by adding more objects. However, this method of selecting an active tool has one advantage: there is only a single object. Nothing can be occluded by previously created objects, the entire space is available. By introducing a new scene mode, this feature could be preserved. The scene mode would only have the ability to select existing objects and place them anywhere in the scene. If an object is to be modified, the application will switch back to the appropriate tool mode. The object will be enlarged to allow easy manipulation while hiding the remaining scene. As soon as this manipulation is complete, the application will switch back to the scene mode.

- create objects independently using modelling tools
- store objects in a library
- switch to *scene mode*
- place objects in the scene
- translate, rotate and scale objects

In case this way of separating the scene from the individual objects proves to be too compliated, an alternative method has to be devised. It might be more intuitive to edit objects in place. However, if an object becomes too small, changing its shape will require very precise input handling.

5.2.2 Improve Mesh Data Structure

Currently, the data structure of the mesh cannot be used to its full potential. Even though it has the capability to represent sharp as well as smooth edges, no modelling tool currently makes use of it. As all blob based tools only allow the user to create smooth dents and bumps, it is impossible to create sharp edges. Therefore the resulting objects are mostly organic and have a smooth surface. The only time sharp edges are set is when importing an arbitrary mesh. However, by modifying this mesh, the edges will eventually lose their sharpness. Using the current tools, there simply is no way to change this behaviour. In any case, as blobs are expected to be smooth, adding the ability to create sharp edges seems unnecessary.
A new and different kind of tool could be designed specifically for man-made objects. Such objects mostly consist of sharp edges and flat surfaces. A technique similar to *iWires* [GSMCO09] could be used to preserve sharp edges, and consequently the whole shape. This technique is completely different from anything that is currently implemented. The mesh data structure provides all the operations and parameters required. However, this data has to be collected and combined in order to form the so called wires. Hence a special data structure has to be built on top of the mesh data structure. This new data structure only deals with wires and their relations. If one of the wires gets modified, the mesh has to be modified accordingly.
Initialising a wire-based object requires a few additional steps:

- load mesh (*done*)

- detect sharp edges (*done*)
- try to find connected sharp edges and form wires
- combine these individual wires
- find relations and place wires in groups

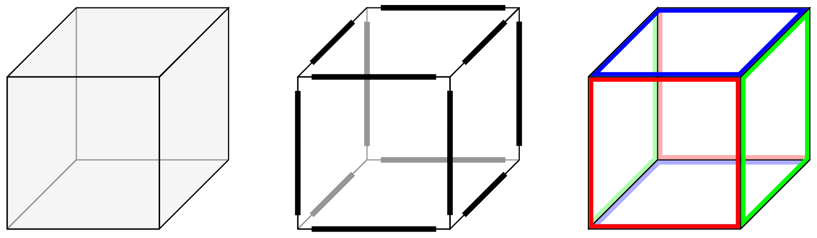


Figure 5.1: Given an arbitrary mesh, the sharp edges have to be detected. Based on these individual edges, wires are formed. If, for example, wires are connected and enclose a common surface, they should be merged. The image on the right shows these finalised wires and their relations. In this case, wires that lie on parallel planes are placed in groups.

Relations are very important. They define the mesh's characteristics. If a wire gets modified, all wires belonging to the same group are influenced. The relations are persistent and therefore preserve the shape of the mesh. The most basic relations are:

- wires are connected
- wires lie on the same plane
- wires lie on parallel planes

As soon as the initialisation procedure is completed, the user can start to modify the object. Instead of operating on a vertex level, wires are selected. These wires have to be modified in a different way and therefore require a new set of operations:

- Translate
- Scale
- Rotate

Wires are modified as a whole, hence their shape remains intact. However, this also limits the possibilities. A cube can never become anything else but an arbitrary cuboid. No new parts can be added, at least not without adding more operations. For example, if a wire is selected, a special operation could allow the user to add a square-shaped wire inside it. This newly created square could then be translated. By repeating this operation multiple times, a cube could eventually be turned into a table by adding legs.
Despite being only a small subset of the functionality provided by iWires, this new tool could be used for interesting applications. However, as it is quite different to the presently available tools, the user interface needs to be adapted. Especially the joystick has to be used in a different way in order to support the new operations.

5.2.3 Improve Appearance

Using the quality parameter, the triangulation of the mesh can be influenced. A higher quality and more triangles will result in a smoother surface with fewer visible edges. However, there are more aspects than just the geometry that influence the appearance. As it was mentioned in previous chapters, the surroundings can be exchanged. Hence a different, more sophisticated environment may be integrated. The only requirement is that the environment is open on one side, because the DAVE only has three sides and not four.

5.2.3.1 Shadow Generation

Shadows can greatly improve the appearance. If an object does not have a shadow, it will seem even more artificial. Creating shadows is a common task in computer graphics. Several techniques have been devised. Their quality as well as complexity can vary greatly. The simplest form is to draw an image on the floor. The image does not necessarily have to resemble the object's shape. This technique has been widely used in computer games due to its simplicity. No extra calculations are necessary. The object's complexity has no influence at all. However, as soon as a shadow gets larger, it becomes obvious that the shadow does not match the object's shape. Therefore the modelling programme uses more advanced ways of drawing shadows. Two different modes are implemented. The first one draws an object a second time using a special transformation matrix. This matrix projects an object onto a plane. By colouring this projection, it will seem shadow-like. The shadows however are limited to a single plane. In case a different object lies on the floor, the shadows will not cover it properly.
An alternative mode utilises shadow volumes in order to create a more convincing appearance and to deal with this problem. This particular implementation is called *depth-fail* and was discovered by William Bilodeau and Michael Songy [BS99] in 1999. Id Software's John Carmack discovered the technique independently, hence it is sometimes referred to as *Carmack's Reverse*. Using the position of the light source, the occluded area can be calculated. This area is covered by a shadow volume that is generated by extending the outer edges of a mesh. Utilising the graphics card's *stencil buffer* and a sequence of operations, very realistic shadows can be created. However, most of the calculations have to be done on the central processing unit. Additionally, these shadows are very precise and have no smooth transitions unlike shadows in the real world.
Either one of these techniques can be activated. By default, the projected shadows are selected in order to improve the performance. However, these shadows are only cast on the floor. Neither the surroundings nor the object itself is influenced. Shadow volumes give a superior result. Both of these techniques are considered old fashioned nowadays. By implementing a third method, the shadow quality can be improved without severely decreasing the application's speed. This new method is called *shadow mapping*. It was proposed by Lance Williams [Wil78] more than thirty years ago. The resulting shadows are not be as precise as shadows generated by the previous methods, even though smooth edges can be generated. Most of the work is done using the graphics hardware. Therefore the calculations can be performed considerably faster. There are two separate steps. First of all the scene has to be rendered from the point of view of the light source. By extracting information from the depth buffer, the shadow map is generated. This shadow map is a normal texture and by changing its resolution, the shadow quality can be altered. Low resolution results in clearly visible artifacts, whereas a high resolution can display any shapes and utilise functions such as anti aliasing to further improve the quality. In the final step, the scene has to be rendered from the camera's point of view. By accessing the shadow map, the shadows can be generated. Smoothing can be accomplished by interpolating between neighbouring values.

5.2.3.2 Different Materials

Presently, different colours can be applied to an object. Depending on the underlying texture, the appearance changes. As this approach is rather limited, a special multi-texture mode was implemented.

Instead of directly changing the colour, different textures are placed on the surface of the object. This method proved to be very powerful for certain applications such as landscape modelling. However, its full potential has not been utilised so far. The textures are drawn using a shader programme. Shaders could be used to do so much more than just blending a few textures. Currently, the shader programme has the ability to activate *normal mapping*. Normal mapping is a technique that changes the surface structure of a mesh by altering the normal vectors for each pixel. These normal vectors are used to calculate the lighting. Hence a surface can appear to have bumps even though it actually is completely flat. The normal map is stored in a texture. The red, green and blue components represent the x, y and z values of a vector.
Normal mapping is just the beginning. Far more effects can be implemented. As a texture is now accompanied by other textures, it will from now on be referred to as a material. Each material can consist of multiple textures. Each texture is used for a different purpose. A short list of possible properties are:

- **Diffuse map**: default texture
- **Normal map**: provides surface structure
- **Displacement map**: provides illusion of depth
- **Specular map**: defines which parts of the material are shiny
- **Transparency**: a single value that defines how transparent the material is
- **Reflection**: activates reflection mapping

A specular map can greatly influence the appearance of a material. The brightest spot of light, the specular highlight, is visible on shiny surfaces. By using a specular map, a surface does not have to be shiny everywhere. Parts of it can be matt as well. This becomes particularly helpful if, for example, a metal surface is partly covered by rust. Rust does not reflect as much light as ordinary metal. Furthermore this allows materials to be mixed. A wooden surface for example can now contain shiny metal parts.
In case the normal map does not provide enough surface structure, the illusion of depth can be increased by using a displacement mapping technique. A special map defines how far the point of a surface has to be displaced. Several different techniques exist. Most of them only perform virtual displacement, such as the popular parallax mapping. Depending on the viewing angle, the appearance changes. Hence a nail on a wooden surface can appear to stick out.
The remaining two parameters, transparency and reflection, can turn any material into something glass-like. Reflections can be approximated by using spherical environment mapping. Transparency, however, is very difficult to implement. If a small part of the surface is turned into glass, it is then possible to look *inside* of an object. What is a user going to see? Just the back of the opposite side? This problem could be avoided by turning transparency into a global property. Either the whole object is transparent or opaque.
Besides changing the appearance, materials can also behave differently. If constraints like for instance the flexibility are defined, choosing the material can influence the resulting shape. A very flexible material might lead to smooth surfaces whereas a stiff material will produce many edges. The landscape modelling tool starts off with a flat surface. If this surface is made out of steel, creating large hills will require a great deal of work. Only small dents can be formed. In order to create a hill, many small dents will have to be combined.

5.2.4 Loading and Saving of Meshes

The mesh data structure already has the ability to save meshes into files. Presently the PLY file format is preferred, as it supports all the required properties. Morever, due to its simplicity, many modelling

programmes can read it. Hence it is a convenient way of storing meshes. Blob based modelling tools are initialised using one of these PLY files. Therefore they already possess the ability to both load and save arbitrary meshes. The snake modelling tool however requires a special kind of mesh. Circles are extruded in order to form pipe like shapes. Such circles consist of a certain number of vertices and edges. The number depends on the mesh quality parameter. This parameter has got to stay the same throughout the creation process. Thus, loading a previously generated mesh can lead to complications. Old and new pipes might not be able to merge, or at least not in the way they usually do. The stored mesh will have to be analysed and individual pipes detected. This preprocessing step is currently missing. It probably is more difficult to implement this step rather than converting the whole tool to be blob-based. In case it is based on the blob tool, storing information about individual circles becomes obsolete. Unfortunately the current implementation still has some advantages with respect to the resulting mesh quality and appearance and it will probably not be replaced in the near future. Consequently, the user currently only has the option to save and load blob-based meshes.
In case these problems have been solved and all modelling tools have the ability to load meshes, the user interface will have to be adapted. It should be easy to load and save meshes. Saving is executed by pressing the appropriate icon. As the user does not have access to a keyboard, the filename is chosen by the programme. A combination of the current date as well as the name of the modelling tool are sufficient to get a unique filename. A picture of the mesh is stored alongside the PLY file. Therefore one does not have to load the entire mesh in order to identify it. A quick look at the picture will do the job. Currently, the last five blobs and landscapes are displayed on the right side as flat white cuboids showing these pictures. By clicking on one of these cuboids, the corresponding mesh will be loaded. It will be necessary to change the user interface in case this small selection of stored meshes turns out to be insufficient.
The mesh data structure tries to maintain a regular triangulation. This triangulation is necessary in order to allow the user to modify arbitrary parts of a mesh. However, not all triangles and vertices are required to display the final mesh. A flat surface, for instance, can be represented by just a few triangles in contrast to a bumpy surface that contains many little features. As a result, the mesh can be simplified before it is saved. The filesize will be reduced as well as the mesh's complexity. Preliminary tests using Blender's *Decimate modifier* [Blea] have shown great potential. However, only the geometry is taken into account. Features like vertex colours are ignored and damaged beyond repair. A mesh simplification algorithm will have to respect all properties and not only the geometry.

5.2.5 Performance

An application can always be optimised in order to make it faster. Very often, parts of an algorithm can be skipped without changing the outcome. This however requires a thorough understanding of the functionality. By removing supposedly unused parts, the programme can be damaged and new errors are introduced. Furthermore, optimisations should be performed when the programme is more or less complete. Just because a certain feature is not needed at an early stage of development, it does not mean that it will not be required later on. Having to find and repair such *optimisations* can be very time consuming.
The modelling programme cannot be considered as optimised. The calculations as well as the rendering process are far from being efficient. In order to maintain the mesh consistency, checks are performed prior to and after each operation. Errors often lead to incorrect data that eventually made the programme crash. Now, these errors have been fixed. Hence some of the checks are now obsolete. Finding these checks is a complicated task. Each and every exception has to be taken into account. Assumptions like 'well, this can never happen' have to be validated. Hence the programme must be tested continuously. By devising unit tests, all aspects could be examined automatically.
The rendering process currently supports two different modes. The default mode iterates through the mesh data structure and builds the geometry on-the-fly. This has to be done each time it is rendered. However, the geometry only changes if the user performs a mesh modification. Therefore the geometry

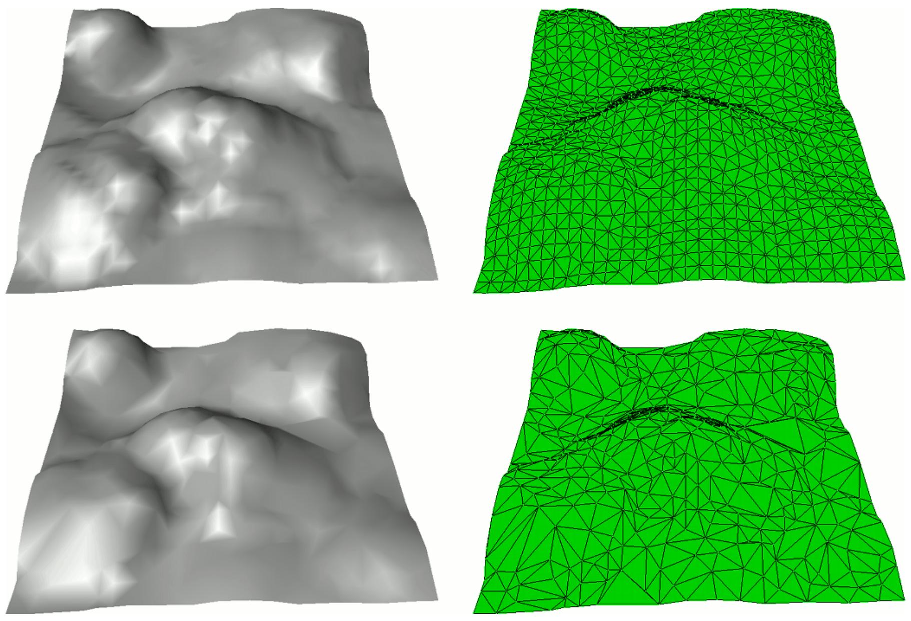


Figure 5.2: This mesh was generated by the landscape modelling tool. The original version on the top consists of 931 vertices and 1740 faces. Applying Blender's decimate modifier with a ratio of 0.4 results in only 380 vertices and 695 faces. The complexity was reduced quite significantly. However, the difference is clearly visible when using per-vertex lighting. As per-vertex lighting does not work particularly well for either model, the modelling application uses per-pixel lighting instead.

stays the same most of the time. Doing the same calculations over and over again seems pointless. Fortunately, OpenGL supports various features that can be used to speed up the process. The most important one is called *vertex buffer objects* (VBO). The geometry has to be defined once and is stored in the memory of the graphics card. By using a unique identifier, the VBO can be accessed. If the applications want to draw it, the identifier has to be passed to the graphic card. Hence the central processing unit no longer has to perform these complicated calculations constantly.
VBO can already be utilised by the programme. However, despite being quicker than the default mode most of the time, there still are some unsolved issues. Transferring the geometry to the graphics card and building the VBO takes a considerable amount of time. If, for example, the landscape modelling tool is active and the user is creating a hill by grabbing a part and dragging it up, the mesh changes all the time. Hence the VBO has to be completely rebuilt for almost every frame. The performance will therefore be significantly worse. As soon as the user is finished with the modification, the frame rate will go up again. The following two items could solve this problem:

- only perform mesh optimisations if the modification is complete
- try to update the existing VBO instead of completely rebuilding it

Updating the VBO could lead to significant speed-ups, especially when colouring the mesh. Changing the colour does not effect the geometry. Hence only the colour property needs to be updated. Changing the mesh optimisation behaviour however has an influence on the geometry. If the optimisation is performed only once rather than continuously, the result will be different. An additional post-processing step will be necessary.
Alternatively, the mesh could be split up into separate parts for the rendering process. Therefore not the entire mesh has to be updated if some part is modified.

5.3 Summary

Writing an application for the DAVE has definitely been a very interesting experience. This application has very special requirements, such as utilising the tracking system and controlling multiple projectors. New features were added to the application and had to be validated in the DAVE. Displaying the same object on a regular screen or inside the DAVE can lead to completely different results. An object that seems to be of the right size on a monitor turns out to be too large in the DAVE. By introducing the DAVE simulator, such a problem can be detected immediately without having to use the real DAVE. Nevertheless, certain aspects of an application, for example input handling, can only be simulated to a certain degree. While keys on a keyboard can be mapped to replace buttons on the joystick, simulating the movements of the user as well as the joystick remains a difficult task. In order to look at an object from a different angle in the DAVE, the user has to walk a few steps. However, these steps usually involve a translation as well as a rotation. If the user wants to repeat the same movement using the simulator, multiple keys have to be pressed. It requires a great deal of experience to be able to do this. Hence the user needs experience to use the simulator properly. Nevertheless, the application developed in this thesis turned out to be very usable, with or without the simulator.
Consequently, the programming effort required to implement this application was extremely extensive. Accessing input events or creating sound effects is accomplished by using external libraries. However, the graphics part was written from scratch. OpenGL functions are called directly without the help of external libraries. Irrlicht or any other freely available open source 3D graphics engine could have been used instead. These engines usually only require a set of vertices and faces. This data can be provided by the mesh data structure. However, for the sake of simplicity and desire to gain as much knowledge about computer graphics as possible, no graphics engine was used. This choice resulted in extra work and gave the opportunity to learn new and exciting aspects of OpenGL.

Bibliography

[BHZK05] Botsch M., Hornung A., Zwicker M., Kobbelt L.: High-quality surface splatting on today's gpus. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics 0* (2005), 17–141. 14

[Blea] Blender Foundation: *Blender 2.49 Manual - Decimate Modifier.* http://wiki.blender.org/index.php/Doc:Manual/Modifiers/Mesh/Decimate. 73

[Bleb] Blender Foundation: *Blender 2.49 Manual - EdgeSplit Modifier.* http://wiki.blender.org/index.php/Doc:Manual/Modifiers/Mesh/EdgeSplit. 64

[BLRC03] Bruno F., Luchi M. L., Rizzuti S., Calabria U. D.: The over-sketching technique for free-hand shape modelling in virtual reality. In *In Proceedings of Virtual Concept 2003, Biarritz* (2003), pp. 5–7. 9

[Boua] Bourke P.: *PLY - Polygon File Format, also known as the Stanford Triangle Format.* http://local.wasp.uwa.edu.au/˜pbourke/dataformats/ply/. 44

[Boub] Bourke P.: *Wavefront .obj file format specification for the Advanced Visualizer software.* http://local.wasp.uwa.edu.au/˜pbourke/dataformats/obj/. 43

[BRF01] Bandyopadhyay D., Raskar R., Fuchs H.: Dynamic shader lamps: Painting on movable objects. In *In Proceedings of Int. Symp. On Augmented Reality* (2001), pp. 207–216. 8

[BS91] Bloomenthal J., Shoemake K.: Convolution surfaces. *SIGGRAPH Comput. Graph. 25*, 4 (1991), 251–256. 15

[BS99] Bilodeau W., Songy M.: Real time shadows. In *Creative Labs Sponsored Game Developer Conference* (1999), Creative Labs Inc. 71

[BSLM05] Baxter B., Scheib V., Lin M. C., Manocha D.: Dab: interactive haptic painting with 3d virtual brushes. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 180. 8

[BWK02] Botsch M., Wiratanaya A., Kobbelt L.: Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 53–64. 13

[Cat74] Catmull E. E.: *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. 48

[CB04] Cao X., Balakrishnan R.: Visionwand: interaction techniques for large displays using a passive wand tracked in 3d. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 729–729. 11

[CNSD93] Cruz-Neira C., Sandin D. J., DeFanti T. A.: Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM, pp. 135–142. 2

[DG96] Deutsch P., Gailly J.-L.: *ZLIB Compressed Data Format Specification version 3.3.* United States, 1996. 53

[DWS*97] Duchaineau M., Wolinsky M., Sigeti D. E., Miller M. C., Aldrich C., Mineev-Weinstein M. B.: Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97* (Los Alamitos, CA, USA, 1997), IEEE Computer Society Press, pp. 81–88. 30

[FDSB04] Fassold H., Danzl R., Schindler K., Bischof H.: Reconstruction of archaeological finds using shape from stereo and shape from shading. In *9 th Computer Vision Winter Workshop (CVWW), February, Piran* (2004), pp. 21–30. 11

[FHH03] Fellner D., Havemann S., Hopp A.: *DAVE - Eine neue Technologie zur preiswerten und hochqualitativen immersiven 3D-Darstellung.* Tech. rep., Institute of Computer Graphics University of Technology, Braunschweig, Germany, 2003. 2, 32

[GBGS05] Grasset R., Boissieux L., Gascuel J. D., Schmalstieg D.: Interactive mediated reality. In *AUIC '05: Proceedings of the Sixth Australasian conference on User interface* (Darlinghurst, Australia, Australia, 2005), Australian Computer Society, Inc., pp. 21–29. 8

[GSMCO09] Gal R., Sorkine O., Mitra N. J., Cohen-Or D.: iwires: an analyze-and-edit approach to shape manipulation. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers* (New York, NY, USA, 2009), ACM, pp. 1–10. 9, 69

[HF05] Havemann S., Fellner D. W.: *Generative mesh modeling.* PhD thesis, TU Braunschweig, 2005. 14

[Hie06] Hiebert G.: *OpenAL 1.1 Specification and Reference.* http://www.openal.org, 2006. 33

[IMT06] Igarashi T., Matsuoka S., Tanaka H.: Teddy: a sketching interface for 3d freeform design. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, p. 11. 9

[KEA03] Keskin C., Erkan A., Akarun L.: Real time hand tracking and 3d gesture recognition for interactive interfaces using hmm. In *In Proceedings of International Conference on Artificial Neural Networks* (2003). 11

[KFM*01] Keefe D. F., Feliz D. A., Moscovich T., Laidlaw D. H., LaViola Jr. J. J.: Cavepainting: a fully immersive 3d artistic medium and interactive experience. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM, pp. 85–93. 8

[KMZ*07] Keefe D. F., Member S., Zeleznik R. C., Laidlaw D. H., Member S.: Drawing on air: Input techniques for controlled 3d line illustration. *IEEE Transactions on Visualization and Computer Graphics* (2007). 8

[LKG*03] Llamas I., Kim B., Gargus J., Rossignac J., Shaw C. D.: Twister: a space-warp operator for the two-handed editing of 3d shapes. *ACM Trans. Graph. 22*, 3 (2003), 663–668. 12

[LLG93] Liang J., Liang O., Green M.: Geometric modeling using six degrees of freedom input devices. In *In 3rd Int'l Conference on CAD and Computer Graphics* (1993), pp. 217–222. 10

[MBA*04] Murayama J., Bougrila L., Akahane Y. K., Hasegawa S., Hirsbrunner B., Sato M.: Spidar g+g: A two-handed haptic interface for bimanual vr interaction. In *Proceedings of EuroHaptics 2004* (2004), pp. 138–146. 11

[Min96] Mine M. R.: *Working in a Virtual World: Interaction Techniques Used in the Chapel Hill Immersive Modeling Program.* Tech. rep., University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1996. 10, 50

[NS03] Nethercote N., Seward J.: Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89*, 2 (2003), 44 – 66. RV '2003, Run-time Verification (Satellite Workshop of CAV '03). 35

[PPW97] Pausch R., Proffitt D., Williams G.: Quantifying immersion in virtual reality. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 13–18. 3

[RLK09] Rost R. J., Licea-Kane B.: *OpenGL Shading Language.* Addison-Wesley Longman, Amsterdam, 2009. 27

[RRT95] Ramey D., Rose L., Tyerman L.: *Lightwave / OBJ material file.* Alias—Wavefront, Inc., http://local.wasp.uwa.edu.au/˜pbourke/dataformats/mtl/, 1995. 43

[RVB02] Reiners D., Vo G., Behr J.: Opensg: Basic concepts. In *In 1. OpenSG Symposium OpenSG* (2002). 32

[SG07] Schou T., Gardner H. J.: A wii remote, a game engine, five sensor bars and a virtual reality theatre. In *OZCHI '07: Proceedings of the 19th Australasian conference on Computer-Human Interaction* (New York, NY, USA, 2007), ACM, pp. 231–234. 11

[She99] Sherstyuk A.: Shape design using convolution surfaces. In *In Proceedings of Shape Modeling International '99* (1999), pp. 56–65. 15

[Sil06] Silicon Graphics, Inc: *glTexGen - control the generation of texture coordinates.* http://www.opengl.org/sdk/docs/man/xhtml/glTexGen.xml, 1991-2006. 63

[SL08] Settgast V., Lancelle M.: *Virtual Reality Lab.* Tech. rep., Computer Graphics and Knowledge Visualization, TU Graz, 2008. 2, 3

[SPHB08] Schlömer T., Poppinga B., Henze N., Boll S.: Gesture recognition with a wii controller. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction* (New York, NY, USA, 2008), ACM, pp. 11–14. 11

[SPS02] Stallman R. M., Pesch R., Shebs S.: *Debugging with GDB : The GNU source-level debugger for GDB version 5.1.1*, 9th ed. GNU Press, 2002. 34

[SS08] Schmidt R., Singh K.: Sketch-based procedural surface modeling and compositing using Surface Trees. *Computer Graphics Forum 27*, 2 (2008), 321–330. Proceedings of Eurographics 2008. 9

[Wil78] Williams L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (1978), 270–274. 71

[ZHH06] Zeleznik R. C., Herndon K. P., Hughes J. F.: Sketch: an interface for sketching 3d scenes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, p. 9. 9

[ZPKG02] Zwicker M., Pauly M., Knoll O., Gross M.: Pointshop 3d: an interactive system for point-based surface editing. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM, pp. 322–329. 14

List of Figures

Index