Lukas Loibnegger

Material and Effects System for Atlas-based Shading

Master Thesis

zur Erlangung des akademischen Grades
Diplom-Ingenieur

eingereicht an der
Graz University of Technology

Betreuer
Markus Steinberger Ass.Prof. Dipl.-Ing. Dr.techn. BSc

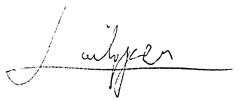
Mitbetreuer
Philip Voglreiter Dipl.-Ing. BSc

Institute of Computer Graphics and Vision

Graz, Dezember 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.



29.11.2019

Date

Signature

Abstract

Modern graphical applications such as games, 3D guided tours or even medical applications strive for a realistic representation of the real world. Virtual Reality has been of interest to achieve this goal for a long time but there are still problems to overcome. This thesis shows techniques that enhance the visual quality of an existing rendering pipeline that tries to eliminate the problem of latency for wireless Virtual Reality solutions. The implementation is concentrated on providing a material system that is versatile, rendering transparent surfaces and mimicking real life phenomenons like fire and smoke with particle systems. The pipeline is able to handle interactive frame rates for rendering different materials and several transparent objects. Still, it turned out that the exact implementation of alpha blending does not fit well with particle rendering whereas other approaches would give better performance whilst keeping a visually pleasing effect.

Contents

List of Figures

1 Introduction

This thesis concentrates on providing three techniques on improving the visual quality of an existing rendering pipeline that tries to divide the concept of a traditional rendering pipeline into a powerful rendering server which sends data wireless to a head-mounted display [Mueller et al. (2018)]. The three techniques are:

- ***Material system:*** Provides an easy interface to change the look of 3D object surfaces
- ***Transparency:*** Enables rendering of transparent objects
- ***Particle systems:*** Provide a system for dynamic effects

2 Problem

When looking at current graphical applications such as in figure 2.1, we encounter high fidelity visuals. Most of the time a virtual world is represented in a distinct style, often quite close to a real world representation. To achieve this close-to-reality representation approximations and simplifications have to be used, since modern hardware is not capable of simulating and presenting physically correct rendering in real time.

Figure 2.1: Graphical application [Unity (2015)].

A lot of new technologies are currently developed to put the user deeper into virtual worlds. One of the most popular techniques is a Virtual Reality headset (see figure 2.2), which gives a person the impression of exploring a virtual environment. These headsets are usually put together with two screens and lenses, which project a left and right image onto the respective eyes of the user. In order to achieve a good experience for users, it is necessary to have a high resolution and refresh rate for each eye. Preferably these headsets should support 4K and 120Hz for both eyes. Current hardware is barely able to render at such numbers for only one screen, so there is still room for improvement.

Figure 2.2: Virtual Reality headset [HTC (2016)]

The aforementioned technique is typically set up with a PC and the Virtual Reality headset connected to the PC with cables, to handle the transferring of the images and tracking data which is used for moving around in the virtual world. A lot of users are not comfortable with a tethered approach for several reasons. It is not only limiting the movement, but also increasing the fear of stumbling over a cable. A solution to this problem is to use a wireless headset which receives data directly from a PC or the usage of a smart-phone (see figure 2.3) which is rendering the images itself [Steed

and Julier (2013)]. In principle this solution is good, especially since smart-phones are housing very high resolution screens these days, but since a full fledged PC is not able to render at those high resolutions a smart-phone is even less suited for that.

Figure 2.3: Phone based VR [Samsung (2015)].

To circumvent this issue the proposed solution is to render to a shading atlas on a high performance PC first, send the data to the phone and finish the rendering on the phone itself. With this approach you do not have the problem of high latency and poor rendering quality [Mueller et al. (2018)]. Some further details will be supplied in section 3.1.

This thesis is going to concentrate on providing a visually pleasing environment for the shading atlas based rendering. To accomplish this task the three techniques described in chapter 1 will be implemented.

2.1 Material System

For simplicity of demonstration we will be using selected scenes as testing environments. Those scenes consist of several parts of information:

- ***Mesh data:*** just plain data points in 3D space that represent the model to be shown
- ***Materials:*** consists of textures, texture scales and property values (e.g.: glossiness)
- ***Object:*** combines mesh data with materials and also adds other information such as position, rotation etc.

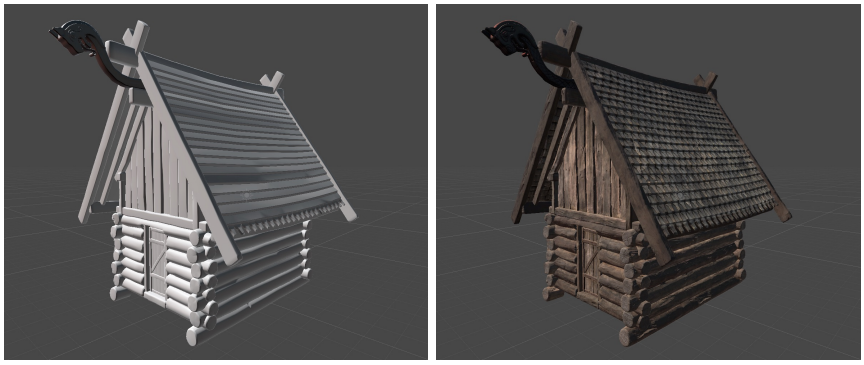


Figure 2.4: Comparison between mesh and applied material

The goal of this task is to define materials through a configuration file and assign them to objects in the scene. The main parts of a material instance are:

- ***Main texture:*** Texture providing the main coloring of the object.
- ***Normal map:*** Texture providing surface details [Krishnamurthy and Levoy (1996)].
- ***Detail textures:*** Consists of three textures that add fine details to the object (detail texture, detail normal map, detail mask).

2.2 Transparency Rendering

Most parts of a scene are opaque which can be handled by rasterization hardware easily, but there are also transparent parts such as windows, foliage and see through materials. Blending is usually done with the following formula:

$$C_{result} = C_{destination} \cdot (1 - alpha_{source}) + C_{source} \cdot alpha_{source}$$

The source alpha value $alpha_{source}$ is used to determine how much the source color C_{source} and destination color $C_{destination}$ contribute to the result color C_{result}. Unfortunately this equation is not associative in regard to the correct ordering when there are more than two objects blended together.

A simple solution would be to sort the 3D objects based on camera position and rotation before rendering. This is feasible for small scenes with simple objects but is computationally too expensive for large scenes. The aim of this thesis is to provide a real time solution for order-independent-transparency as described by Yang et al. (2010) which constructs linked lists per pixel and sorts them. With this approach there is no need for upfront knowledge about the scene.

Figure 2.5: Transparency Rendering. Model by TooManyDemons (2016)

2.3 Particle Systems

Another common approach to mimic real world physical phenomenons and artistic effects is to use a class of fuzzy objects that are simulated individually [Reeves (1983)]. Depending on the configurability of the system a wide variety of effects can be achieved, ranging from fire effects to magic spell effects and even the simple animation of flowing water. The key performance indicator besides the configurability is the number of simultaneous particles that are rendered.

Since this thesis also concentrates on implementing transparency rendering, particle systems will be incorporated in the same pipeline.

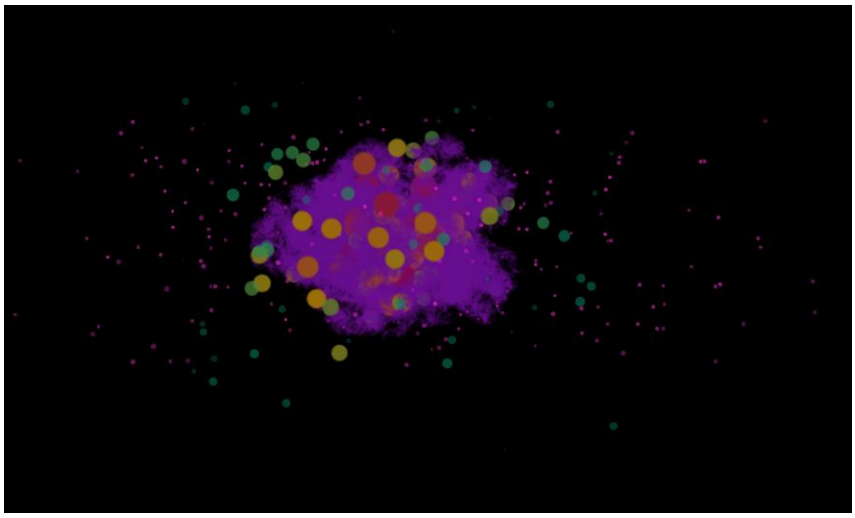


Figure 2.6: Particle effects.

3 Related Work

This thesis builds upon several concepts and methods that are used to find a solution. The main piece of work that needs to be mentioned is Shading Atlas Streaming (SAS) [Mueller et al. (2018)] which is used as a basis for providing the visual enhancements. The enhancement by transparency rendering is based on work by Yang et al. (2010), which describes a method of order-independent transparency rendering by using linked lists. A different approach to transparency rendering was proposed about ten years earlier [Everitt (2001)] where transparency is resolved by ”peeling” layers off the surface. The enhancement by particle systems was proposed by Reeves (1983).

3.1 Shading Atlas Streaming

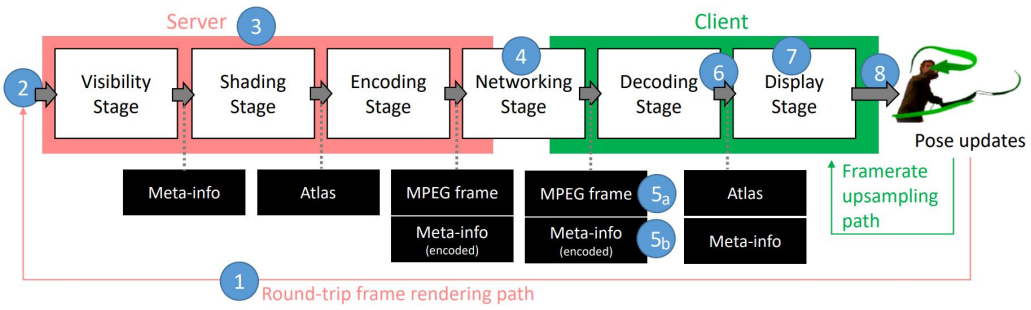


Figure 3.1: Shading Atlas Streaming Pipeline

SAS is a novel object-space solution which tries to solve the problem of streaming VR content to an untethered headset. It can be seen in figure 3.1

that this approach splits the rendering pipeline into a server and client part. The pipeline is executed with the following steps:

1. The current view matrix of the client is sent to the server.
2. The server receives the view matrix.
3. Using the new view matrix the following steps are done:
   a) Visibility Stage: A list of primitives is constructed that were not visible in the last frame.
   b) Shading Stage: New primitives are shaded to the atlas and old ones are updated.
   c) Encoding Stage: The atlas is encoded into a MPEG frame.
4. The collected information (MPEG frames, position information etc.) is sent to the client.
5. MPEG frames are decoded and position updates are written to a vertex buffer.
6. The new information sits idle until the client starts a new frame.
7. The new server frame is rendered.
8. Until a new server frame arrives, existing atlas data are used to perform framerate upsampling.

Figure 3.2: Rendered Scene with corresponding Shading Atlas.

3.2 Transparency Rendering: Depth Peeling

The idea behind depth peeling is to render transparent geometry in layers. This is done by using one render pass for each depth layer that should contribute to the final image. Each layer is stored to a viewport-sized render texture which are combined in a final pass. Assuming that there exist two depth units:

```
for (layer=0; layer<num_layers; layer++)
{
  clearColor;
  firstBuffer = layer % 2;
  secondBuffer = (layer+1) % 2;
  depth unit 0:
    if(layer == 0)
      disable(DEPTH_TEST);
    else
      enable(DEPTH_TEST);
    bind(firstBuffer);
    disable(DEPTH_WRITE);
    depthFunc = GREATER;
  depth unit 1:
    bind(secondBuffer);
    clearDepth();
    enable(DEPTH_WRITE);
    enable(DEPTH_TEST);
    depthFunc = LESS;
  render();
  colors[layer] = colorBuffer;
}
```

Since it is not possible to perform two depth tests in one render pass the paper suggests using the shadow mapping unit which can be exploited as a second depth test. Instead of rendering from the light point of view the camera position and angles of the scene are used. The result of the shadow map comparison is written to the alpha channel which is then used in an alpha test that serves as a ”depth test”.

3.3 Transparency Rendering: Per-Pixel Linked Lists

This paper [Yang et al. (2010)] was released about ten years after the depth peeling approach [Everitt (2001)] and takes advantage of newer GPU hardware features. The introduction of atomic operations on values and textures enabled the implementation of per-pixel linked lists. The idea is to collect a list of fragment candidates for each pixel in a first pass, sort and display them in a second pass. The first pass works as follows:

```
entry_id = atomicAdd(atomic_id_counter, 1);
last_id = imageAtomicExchange(list_header, pixel_coord, entry_id);
linked_list[entry_id].color = color;
linked_list[entry_id].depth = depth;
linked_list[entry_id].last_id = last_id;
```

A unique ID is requested from the atomic ID counter and stored in the list header. While storing the new ID, the previous ID is atomically swapped and stored in the new list entry. The color and depth information is then used in a second pass which is invoked using a fullscreen quad:

```
entry_id = imageLoad(list_header, pixel_coord);
sort(entry_id);
final_color = vec4(0, 0, 0, 1);

while(entry_id != LIST_END)
{
  current_color = linked_list[entry_id].color;
  final_color = mix(final_color, current_color, current_color.a);
  entry_id = linked_list[entry_id].last_id;
}

return final_color;
```

The last written ID is loaded from the header and the corresponding list entries are sorted based on the depth information. A final color is calculated by traversing the list from front to back and blending the colors together.

3.4 Particle Systems

This paper [Reeves (1983)] describes systems of particles that resemble dynamic fuzzy objects. They differentiate from 3D models in three characteristics:

- Instead of a fixed surface, models are represented by a set of simple primitives that define their volume.
- Instead of static models, particle systems are highly dynamic.
- Instead of a deterministic model, particle systems are controlled by random variables.

Particles are ”born” and after a defined lifetime they ”die”. Their movement and appearance are controlled over their lifetime by random variables, but still limited by parameters that define the overall shape of the particle system. The paper depicts the following useful parameters for controlling particle systems:

- Number of new particles over period of time.
- Generation shape: defines initial position of particles
- Velocity
- Size
- Color
- Transparency
- Lifetime

4 Method

State of the art VR systems rely on image-based rendering (IBR) techniques to hide latency between head pose updates and the corresponding frame being displayed on the VR device. The most prominent solution is Time Warping [Mark, McMillan, and Bishop (1997)]. This solution extends the rendering viewport by a small amount, renders the scene to a texture and until the next frame is ready, performs image warping based on the most current head pose. Although this solution is straight forward it does not lend itself for efficient MPEG encoding due to the larger image size. It also has some perceivable image quality issues.

4.1 Transparency Rendering

The main problem with IBR methods are disocclusion artifacts which occur when the head pose translates in the x or y direction. In those scenarios it can easily happen that geometry gets visible which has not been rendered to the current texture. Implementing transparency as described in chapter 3.2 or 3.3 would not differ from a standard rendering pipeline, but the amount of disocclusion artifacts would increase when transparent geometry has been occluded by opaque geometry and the effect would be even more visible.

Since SAS is an object-space oriented solution it can mitigate disocclusion artifacts by supplying a Potentially Visible Set (PVS) which is constructed by predicting future movements. This set is only marginally larger than the set containing geometry that is visible in the moment when rendering starts. The PVS is constructed during the visibility stage as described in chapter 3.1. In order to include transparent geometry this visibility stage

needs to be extended. In order to correctly display transparent geometry on the client one of the proposed solutions from chapter 3.2 or 3.3 needs to be implemented.

4.2 Particle Systems

Most particle effects like fire and smoke rely on transparency to provide a convincing real life effect. State of the art VR systems using IBR methods will have the same disocclusion problems with particles similar to transparent objects. Integrating a particle render pipeline to an IBR pipeline can be done by using the techniques described in chapter 3.4.

In contrast, an object-based solution like SAS boasts several difficulties when incorporating a particle pipeline. Since particles are highly dynamic not only concerning position and rotation but also concerning texturing, a performant way of transmitting texturing information is needed. There are three approaches proposed to solve this problem:

- ***Naive approach:*** Texturing information for each particle is transmitted individually and independently of texturing information of any other particle.
- ***Texture merging:*** Particles with the same texture in the current frame share the same texturing information which is only transmitted once.
- ***Per-system billboard:*** Each particle system is rendered to a large billboard which is then transmitted as a whole combined with a depth texture for resolving transparency ordering.

5 Solution

In order to start with the implementation of the material system the first step is to get familiar with the programming environment and the used Application Programming Interfaces (APIs). Graphics API Vulkan is the most prominent to serve as an interface between the CPU and the GPU.

5.1 Graphics API: Vulkan

For the past years the main graphics API for scientific applications was OpenGL. The use of DirectX is often not possible since it is only available on Windows. OpenGL is relatively simple to use and allows to draw objects with very little code. The downside is the lack of control over the hardware and therefore a performance decrease [Lujan et al. (2019)]. Vulkan is eliminating this performance decrease by enabling more control over the hardware which helps to fine tune every detail of the rendering pipeline.

5.2 Material System

In order to provide a simple interface for defining materials the starting point is to adapt the existing configuration system for scenes. A material instance consists of one main texture that provides coloring of the 3D mesh, one optional normal map texture that gives surface details and three optional detail textures that add further details to color and surface normal data. The mixing factor for those detail textures can be defined with a detail mask that stores the mixing factor per pixel.

Since some of the textures are optional, the shaders need a simple way to toggle the textures. This is implemented by using specialization constants that provide a default value and can be changed when the shader is loaded with Vulkan. Depending on these constants either the textures are sampled or default values are used (e.g.: the normals loaded from the mesh).

The Blinn-Phong model [Blinn (1977)] is used as shading model which some additional variables are defined for. These mainly consist of information about the light (position, color, radius) but also information about the glossiness of the material and its specular color.

For normal mapping the normals sampled from the normal map need to be converted from tangent space to world space. By the nature of creating normal maps most of the normals will point in the positive z-direction. As can be seen in figure 5.1 the blue tint of the normal map indicates the dominance of the z-direction. This works for geometry that is oriented in such a way that the surface normal points in that direction too, but for all other cases the sampled normals need to be adjusted to the surface normals of the mesh. This is done by converting from the tangent space which is local to the normal maps vectors to world space. In order to convert the normals the TBN matrix is used [Vries (2019)]. It is constructed from the tangent, bitangent and normal vectors of a vertex.

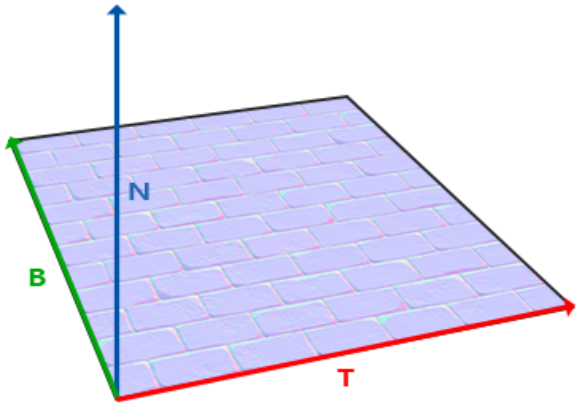


Figure 5.1: TBN Vectors [Vries (2019)]

In figure 5.1 the 3D mesh is a simple plane with N being the surface normal. The tangent vector and bitangent vector (T and B) align with the direction of texture coordinates which is used to compute the vectors.

This simple example has only one surface normal which means that only one pair of tangent and bitangent needs to be computed but for a more complex mesh tangents and bitangents are computed for each vertex and uploaded later to the GPU where they are used in the shader to construct the TBN matrix as follows:

```
vec3 N = normalize(normal_matrix * normal);
vec3 T = normalize(normal_matrix * tangent.xyz);
vec3 B = normalize(cross(N, T)) * tangent.w;
mat3 TBN = mat3(T, B, N);
```

The normal and tangent vectors are vertex inputs in object space and thus need to be converted to world space with the `normal_matrix`. The bitangent vector computation can be moved to the shader since all three vectors are perpendicular to each other which is expressable with the cross product.

The final normal vector is computed as follows:

```
vec3 normal = texture(normal_map, texture_coordinates).rgb;
normal = normalize(normal * 2.0 - 1.0);
normal = normalize(TBN * normal);
```

Since all components of normal vectors need to be in the range -1.0 to 1.0 they are converted from the sampled normal of the texture which is in the range 0.0 to 1.0.

5.3 Transparency Rendering

As already mentioned in chapter 3 there are two possible solutions to order-independent transparency. There exists a newer version of depth peeling [Bavoil and Myers (2008)] which improves the original version by peeling two layers in one pass. Still, per-pixel linked lists are preferred for three reasons:

- ***Render passes:*** Depth peeling requires at least four render passes for achieving transparency with eight layers. The construction and resolving of linked lists can be done in only two passes for an arbitrary number of layers.
- ***Memory consumption:*** For each depth peel one color texture with viewport size is needed even if transparent geometry is only covering a small part of the screen. Linked lists can be constructed with a smaller memory footprint since only actual transparent fragments are written to the list.
- ***Pipeline conformance:*** The linked list approach can be easily adapted to work with Shading Atlas Streaming which will be shown in this chapter.

5.3.1 Overview

In order to split the traditional rendering pipeline into two pipelines that run individually on a server and a client two basic building parts are used:

- ***Patches:*** Geometry is divided into small patches of one, two or three triangles when loading a scene. Each visible patch gets a block of the shading atlas assigned.
- ***Atlas:*** The shading atlas is a texture that stores all the necessary color information for all visible patches. Those patches are regularly subdivided into square superblocks of the same size which are then further subdivided into blocks.

Using those building blocks, the four most important stages complete the server rendering pipeline:

- ***Visibility Stage:*** This is a simple rasterization stage that writes primitive IDs to a texture. Intrinsic to this procedure depth buffering discards invisible primitives. Each primitive ID is linked to a patch which is then marked as visible in the current frame.
- ***Level Selection Stage:*** For each visible patch the level (size of the block) is computed, based on the screen space size and 3D positions of the patch are sent to the client.
- ***Shading Geometry Stage:*** This stage builds a vertex buffer for the shading stage based on visible patches and their selected level.
- ***Shading Stage:*** Uses the afore gathered vertex information to shade patches into the atlas.

Summing up, the server selects the visible patches, deallocates unused space in the atlas, computes the needed space in the atlas, allocates it accordingly, sends geometry information and finally renders color information to the atlas.

On the client side the geometry information is gathered in a global vertex buffer with vertex positions in world space. Each vertex has a texture coordinate assigned which is then used in the fragment shader to sample from the atlas texture. When the atlas is full, for each patch that was not able to fit into the atlas, a color message is sent instead. This color is then used for the whole patch instead of sampling from the atlas.

In order to enable transparent rendering, order independent transparency is implemented using per-fragment linked lists. This approach collects the colors for a given fragment in a linked list using atomic operations, sorts the list in a second render pass based on the recorded depth information and resolves the color for each pixel. The primary advantage is that it is not necessary to sort geometry data before rendering, but performance can vary based on the depth complexity. The buffer space for the linked list is a point of consideration, too.

5.3.2 Implementation

The alpha channel is not handled correctly in the pipeline and ignored in some places. For the client side to handle transparency correctly it has to be adapted to support RGBA colors instead of only RGB.

As described in section 5.3.1, a visibility stage is used to determine which patches need to be shaded later on. This is the first stage that needs altering since patches behind transparent patches are not set visible yet. This functionality is achieved by keeping the existing rendering pass that records opaque primitive IDs which is only executed for opaque materials. The second pass runs for both transparent and opaque materials. In the fragment shader opaque triangles are always marked visible whereas transparent triangles are only marked visible when they are not behind opaque triangles. The rest of the pipeline stays the same on the server side because transparent patches are now handled the same way as opaque ones.

On the client side the atlas mapping stage is rewritten. A linked list is created utilizing an atomic counter that creates an ID for every fragment that corresponds to an entry in the linked list. This entry saves the color, depth and the last entry ID. The current entry ID is saved in a texture storage buffer. In a second resolve pass for each pixel on the rendering surface this texture is read on the corresponding pixel. This gives the starting ID for the pixel. Then the list for each pixel is sorted and the color is resolved by using the following shader code:

```
out_color = mix(out_color, entry.color, entry.color.a);
```

The transparency rendering for the shading atlas renderer mainly consists of an addition to the visibility pass and the atlas mapping stage on the client side. Further additions consist of altering the configuration to have an alpha mode added for materials that can either be `OPAQUE`, `BLEND` or `MASK` mode. The `MASK` mode adds an alpha cutoff which discards fragments with an alpha value higher than a given cutoff.

5.4 Particle Systems

5.4.1 Overview

When implementing particle systems, there is the rendering side running on the GPU that is performant and gives pleasing and correct results regarding alpha blending. On the CPU there is the simulation side that handles loading of configurations and simulating the properties of particles. It delivers customizability and also such good performance compared to the GPU side that it does not bottleneck rendering. This boils down to how many particles can be simulated simultaneously before the CPU struggles to deliver particles to the GPU fast enough.

Although particle systems are different to regular models it is still possible to reuse most of the existing pipeline. The fact that one particle is treated the same as a patch in the pipeline was key for not having to rewrite the whole pipeline but alter few parts instead and add only one new stage.

First the rendering of the particles is done with a naive approach in mind and later refined and improved by doing texture merging.

Naive Approach

Each particle is assigned a patch which is then used to render to the shading atlas. Since there are thousands of particles visible in one frame this leads to a high fill rate of the atlas and also diminishes temporal coherence quite substantially.

Texture Merging

This approach tries to reduce rendering to the shading atlas by grouping together particles with the same texture. A lot of effects use a texture sheet, as seen in figure 5.2, containing several images for the different lifetime stages of a particle. When there are several hundreds of particles alive that use the same texture sheet the possibility that some of them are currently

at the same lifetime stage and therefore have the same texture is very high. Using this knowledge those particles are grouped together and only one of them is actually rendered into the atlas. The atlas information is then shared between all of them.

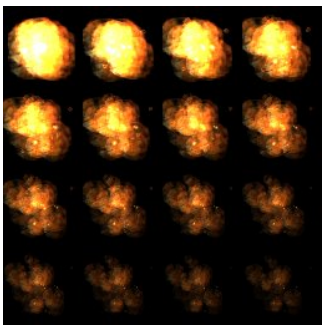


Figure 5.2: Texture Sheet [Murphy (2019)]

Instead of rendering individual particles to the atlas another solution would be to render the particle system as a whole to the atlas and display it on the client as a single billboard. This would require a separate depth texture on the client to resolve alpha blending with static geometry. The benefits would be a smaller footprint in the atlas depending on the view distance of particle systems. This solution was rejected in favour of the current solution due to a problem depicted in figure 5.3, where particles lie behind and in front of a transparent cow. In this case the single depth texture is not able to hold enough information to correctly resolve the particles.

Figure 5.3: Single Billboard Problem

5.4.2 Configuration

In order to provide a wide variety of possibilities on how the particle systems look and how they behave during their lifetime a lot of configuration options are needed. The options are divided into several parts. First there is the common part that handles more general parameters like lifetime, speed and emission rates. Most of these values may be randomized to provide more convincing real life effects. The second part consists of several optional modules that are more concentrated on the behaviour during the lifetime like velocity over lifetime or force over lifetime.

Common Values		
property	format	description
position	float3	world coordinate
rotation	float3	rotation in euler angles
duration	float	duration of emission
looping	bool	emit particles indefinitely
prewarm	bool	starts system in a state like it already ran for one cycle
start delay	float	delay emission
start lifetime	float	lifetime of a single particle
start speed	float	speed of a particle
start size	float	size of the rendered particle
start rotation	float	rotation of the particle
start color	float4	color multiplied onto the sampled texture
gravity modifier	float	gravity multiplier
max particles	int	how many particles can live concurrently in the system
rate over time	float	emission rate per second
texture	string	texture sheet file
tiles	uint2	dimensions of the texture sheet
alpha from grayscale	bool	calculate alpha channel from grayscale

Shape Module Common Values		
property	format	description
type	string	type of shape
position	float3	position offset
rotation	float3	rotation offset
scale	float3	scales size of shape
randomize direction	float	randomizes direction of emission
randomize position	float	offset position by random amount in emission direction
Specific to sphere and hemisphere		
radius	float	radius of the emission shape
radius thickness	float	defines how much of the volume is used for emission
Specific to cone		
angle	float	angle of the cone
arc	float	define the portion of the radius that is used
emit from base	bool	define whether to emit from base or shell
length	float	length of the cone
radius	float	radius of the cone
radius thickness	float	defines how much of the volume is used for emission
Specific to circle		
arc	float	define the portion of the radius that is used
radius	float	radius of circle
radius thickness	float	defines how much of the volume is used
Specific to box		
emit from	string	edge, shell or volume
Specific to edge		
radius	float	length of the edge

Velocity over Lifetime Module		
property	format	description
x, y, z	float3	velocity over lifetime
speed modifier	float	multiplied onto velocity

Force over Lifetime Module		
property	format	description
x, y, z	float3	force over lifetime
randomize	bool	define whether force should randomize each frame or only once

Size over Lifetime Module		
property	format	description
start	float	start size
end	float	end size

Rotation over Lifetime Module		
property	format	description
angular velocity	float	rotational velocity

Color over Lifetime Module		
property	format	description
gradient	Gradient	gradient over lifetime

Color over Speed Module		
property	format	description
gradient	Gradient	gradient over speed
speed range	float2	start and end speed

Example

The following example configuration represents a fire effect that is shown in figure 5.4. A random rotation is applied between -15 and 15 degrees. Using the size parameter the particles are scaled down. They are only affected by gravity and emission happens from a single point. A gradient is applied to blend new particles in and blend dying particles out.

Figure 5.4: Simple Fire Effect

```
position = { x = 0.64841 y = 0.1317 z = -10.49812 }
rotation = { x = -0.5564728 y = -0.4578413 z = -0.440519 }
start_speed = { start = 0.0 end = 0.0 }
start_size = { start = 0.26 end = 0.57 }
start_rotation = {start = -15.0 end = 15.0 }
start_color_begin = { x = 1.5 y = 1.5 z = 1.5 w = 1.0 }
start_color_end = { x = 1.5 y = 1.5 z = 1.5 w = 1.0 }
gravity_modifier = { value = -0.08 }
max_particles = 50
rate_over_time = { value = 4.0 }
texture = "media/particle_test/fire2.png"
tiles = { x = 8.0 y = 4.0 }
alpha_from_grayscale = 1
```

```
shape = {
   type = "edge"
   radius = 0.01
}
color_over_lifetime = {
   gradient_0 = {
      fixed = 0
      color0 = { x = 1.0 y = 1.0 z = 1.0 w = 0.0 }
      color2 = { x = 1.0 y = 1.0 z = 1.0 w = 100.0 }
      alpha0 = { x = 0.0 y = 0.0 }
      alpha1 = { x = 1.0 y = 5.0 }
      alpha2 = { x = 1.0 y = 80.0 }
      alpha3 = { x = 0.0 y = 100.0 }
   }
}
size_over_lifetime = {
   start = 0.11
   end = 1.5
}
```

5.4.3 Implementation

The implementation can be split quite clearly into the configuration loading and simulation part on the CPU and the rendering part on the GPU.

Simulation

In order to have an efficient system for simulating and handling thousands of particles the first thing is to implement a memory pool that allocates particles and keeps track of their deletion.

The memory pool has three lists which are used to keep track of particle instances, active and inactive particles. The instance list holds the actual instances and the other two hold pointers to those instances. Whenever a new particle is requested the inactive particles list is checked whether there are still particles allocated but not used, otherwise a new particle is allocated. On destruction of a particle it is released which triggers the removal from the active particles list and the addition to the inactive list. That way memory allocation, which is far more costly than computation time, is reduced to a minimum and the CPU has more time for simulation.

The actual simulation is done by using several classes that build up a hierarchy:

- ***Particle***: The actual particle class that holds all properties that are updated in each frame. It also provides a method to get the data which is uploaded to the GPU.
- ***ParticleSystem***: Holds a list of *Particles* that correspond to this system. It handles emission of new particles with respect to the provided configuration and calls the particles update method. It also serves as an interface for the GPU representation of particle systems and handles the loading of corresponding resources.
- ***ParticleManager***: Holds a list of *ParticleSystems* and handles GPU resources that are globally used for all particle systems.

Instancing

When rendering 3D models a vertex buffer is used that holds information for each vertex of all triangles in the model. When dealing with particles this can be simplified by using instanced rendering because the 3D data is being reduced down to four 2D points that resemble a quad. Instead of creating a vertex buffer that repeats the particle data for each point in the quad four times two buffers are used:

- ***Vertex Buffer***: Buffer containing six vertices that construct two triangles for the quad.
- ***Instance Buffer***: Buffer containing the actual particle data.

The final part is to tell the GPU to use the same instance data for each of the four vertices and change the draw call according to the actual alive particles of each frame. This is done by having a separate buffer that contains information on how many vertices are drawn (in this case four) and how many instances of this vertex data are drawn (number of particles alive in the current frame).

Single Pixel Patches

A key building block that was used for particles are single pixel patches. They are used for situations where no patches can be rendered to the atlas any more due to space constraints. In this case a representing color for the patch is sent to the client instead and used for the whole patch.

Since particles always have a color that is multiplied onto the sampled texture the single pixel messages lend themselves to be used for this color. The exact use varies for the two implementation approaches.

Naive Approach

With the uploaded particle data the transparency pipeline is altered to incorporate particles. For the naive approach the following alterations were made:

- ***Visibility Stage:*** Add new stage that adds a visible patch for each visible particle.
- ***Level Selection Stage:*** Add computation of screen space size of particle patch based on instance data. Force a single pixel patch for particles with no texture.
- ***Shading Stage:*** Addition of code for computing particle vertex positions and writing the color to the shading atlas.

The main work for this approach is to adapt the shading atlas resources (patch data, triangle data etc.) and also adapt the stages to use the instance data for particles instead of vertex data.

Texture Merging

The idea for texture merging is to use a master patch for each possible particle look and reuse this patch for all other occurrences of this look. In order to manage those master patches an array on the GPU is used where the atlas information and the area of the master patch is stored. In order to get the corresponding master patch for a particle an ID is used which is constructed from the texture sheet ID (assigned on loading of the configuration) and the current stage in the texture sheet. The stages are then updated as follows:

- ***Master Patch Selection:*** In order to select a master patch the particle patch with the largest screen space area is selected in order to give the best possible visual quality.
- ***Level Selection Stage:*** Similar to the naive approach a single pixel patch is forced for all particles except the one that was selected as master patch.
- ***Shading Geometry:*** Still constructs the vertex buffer for the shading stage, but stores the correct master patch atlas information for particles.
- ***Shading Stage:*** Sends a color message for each particle containing the particle color and also renders patches selected as master patches to the shading atlas.

Client resolving

The atlas mapping stage on the client stays the same since particles are sent just like normal patches, only the handling of single pixel patches changes to always multiply the received color onto particle patches.

6 Evaluation

The evaluation will be split into evaluation of the performance of transparency rendering and evaluation of the two particle system rendering approaches. Each scene is traversed using a camera path. There exist four measures that are of interest:

Pixels: Number of pixels written to the Atlas per frame.

Structural Similarity (SSIM): Visual similarity between shading atlas rendering and forward rendering as introduced in Zhou Wang et al. (2004).

Frame Time: Time it takes to render one frame.

Mean Squared Layers (MSL): For each pixel the linked list entry count is squared and added up. This value is then averaged over the number of rendered pixels.

For transparency rendering viking village and robot lab [Unity (2018)] are evaluated whereas viking village actually does not contain any transparent objects. This should give a good idea about the performance hit taken by the additional stages and computations. The performance is measured against the master branch where transparency rendering is not implemented.

In order to evaluate the differences between naive particle rendering and the texture merging approach some additional scenes are evaluated that only have particle systems in it.

Testing is done on a PC with the following specification:
CPU: AMD Ryzen 3 1200 Quad-Core @ 3.10 GHz
GPU: Nvidia GeForce GTX 1060 6GB
RAM: 8 GB DDR4

6.1 Transparency Rendering

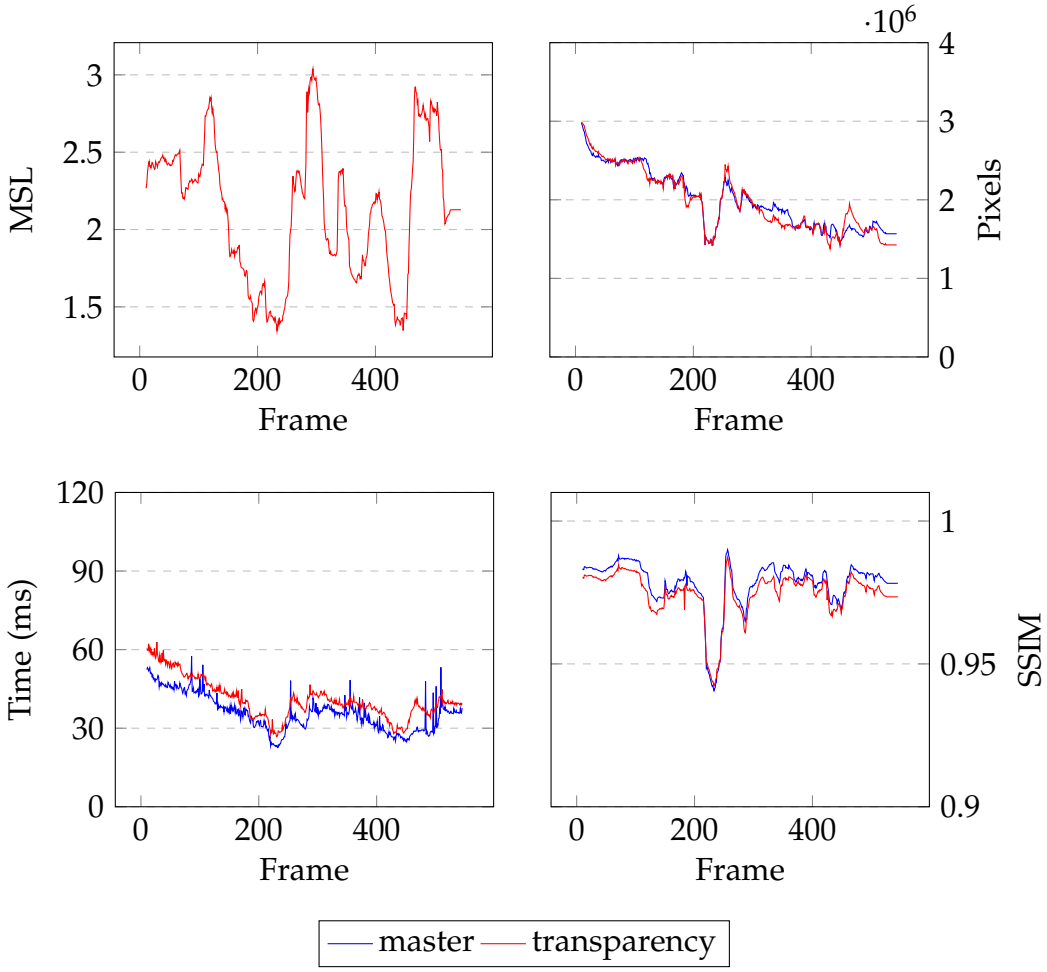


Figure 6.1: Robot Lab.

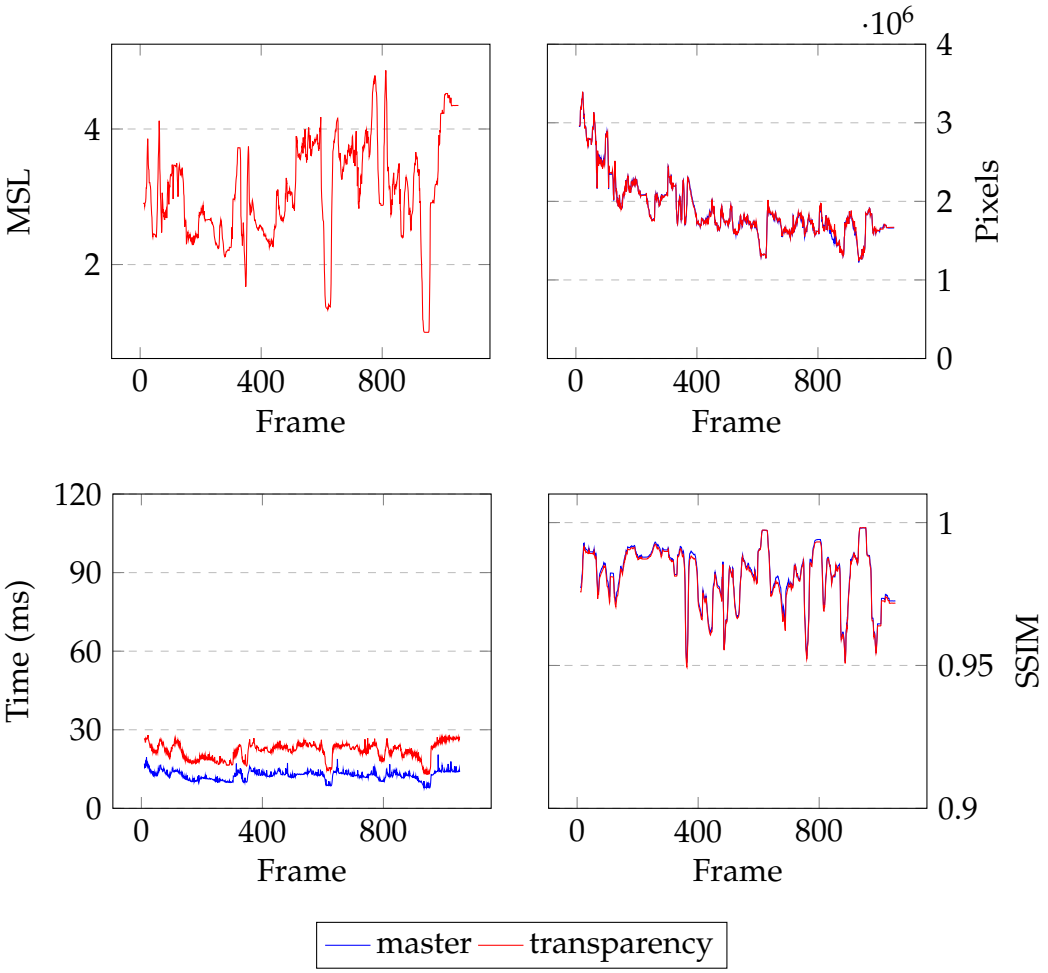


Figure 6.2: Viking Village.

It can be seen in the upper left diagram of figure 6.1 and 6.2 that the MSL does not state clearly how many transparent layers are currently rendered. This is because the visibility pass marks patches visible that are only partially visible. Those patches are then rendered on the client device where no differentiation between transparent and opaque patches is done. Although there is often more than one opaque fragment in the linked list the performance impact is still only seven to ten milliseconds in viking village where no transparent geometry is rendered. The same performance impact applies for robot lab where transparent geometry is rendered.

In viking village the amount of pixels rendered to the shading atlas and the SSIM stay the same compared to rendering without transparency, whereas in robot lab there are more pixels rendered when transparent geometry is visible. The SSIM is only decreased by about 0.5% which can be explained by the higher atlas usage.

6.2 Particle Systems

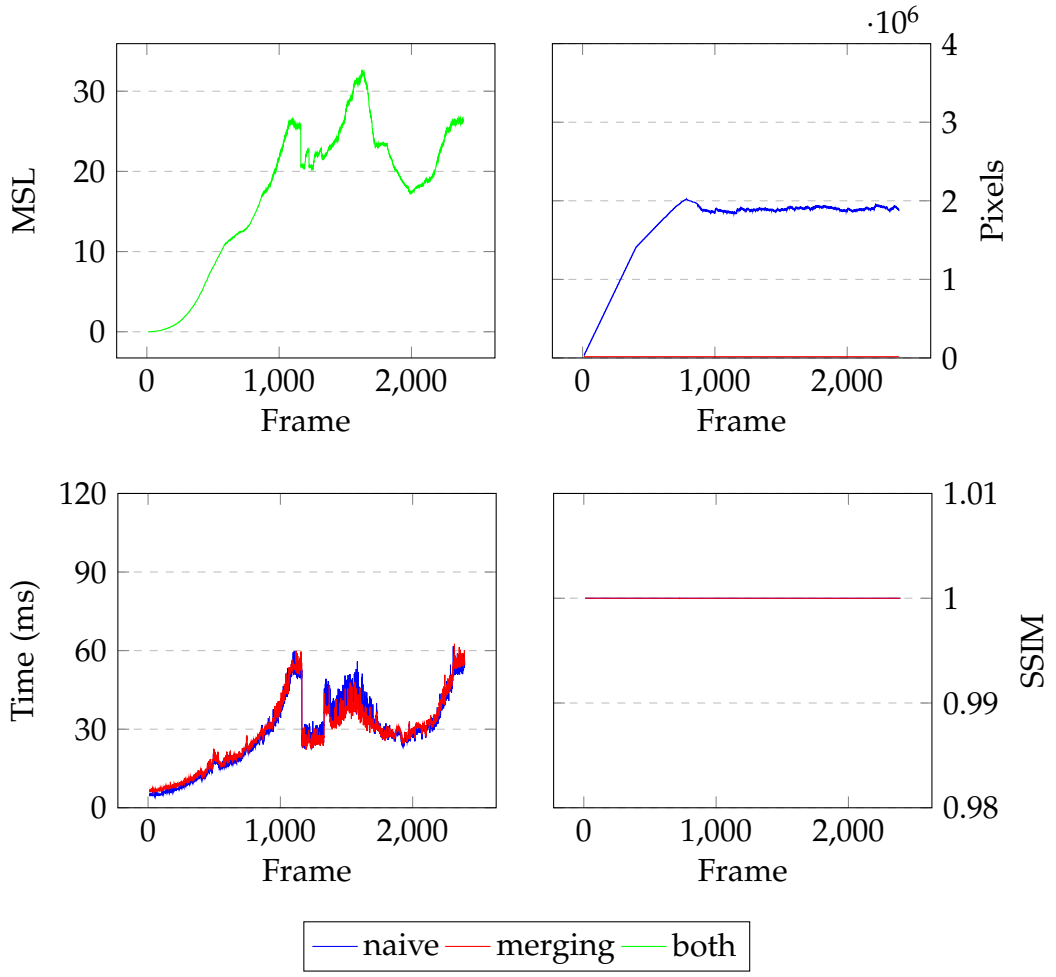


Figure 6.3: This test scene consists of one simple system that has a texture sheet with only one state.

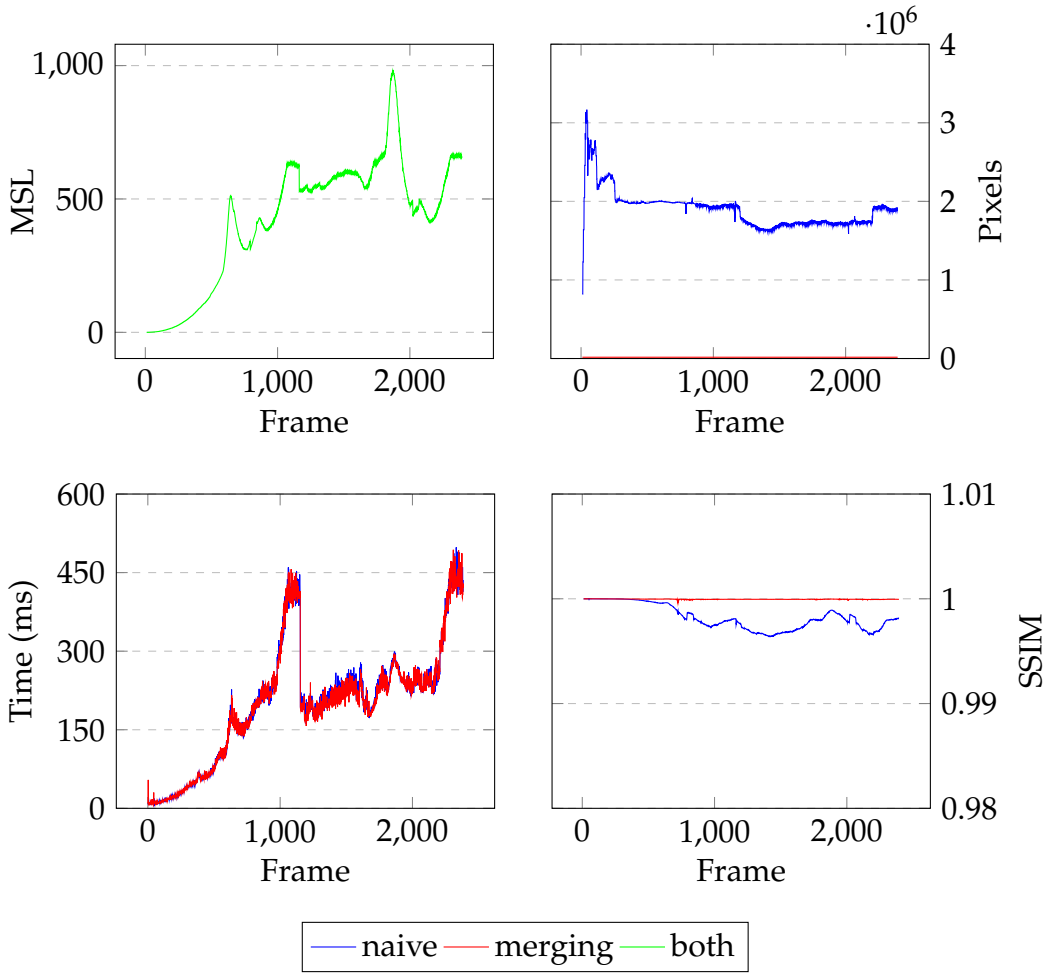


Figure 6.4: This test scene consists of multiple simple systems that all share the same texture.

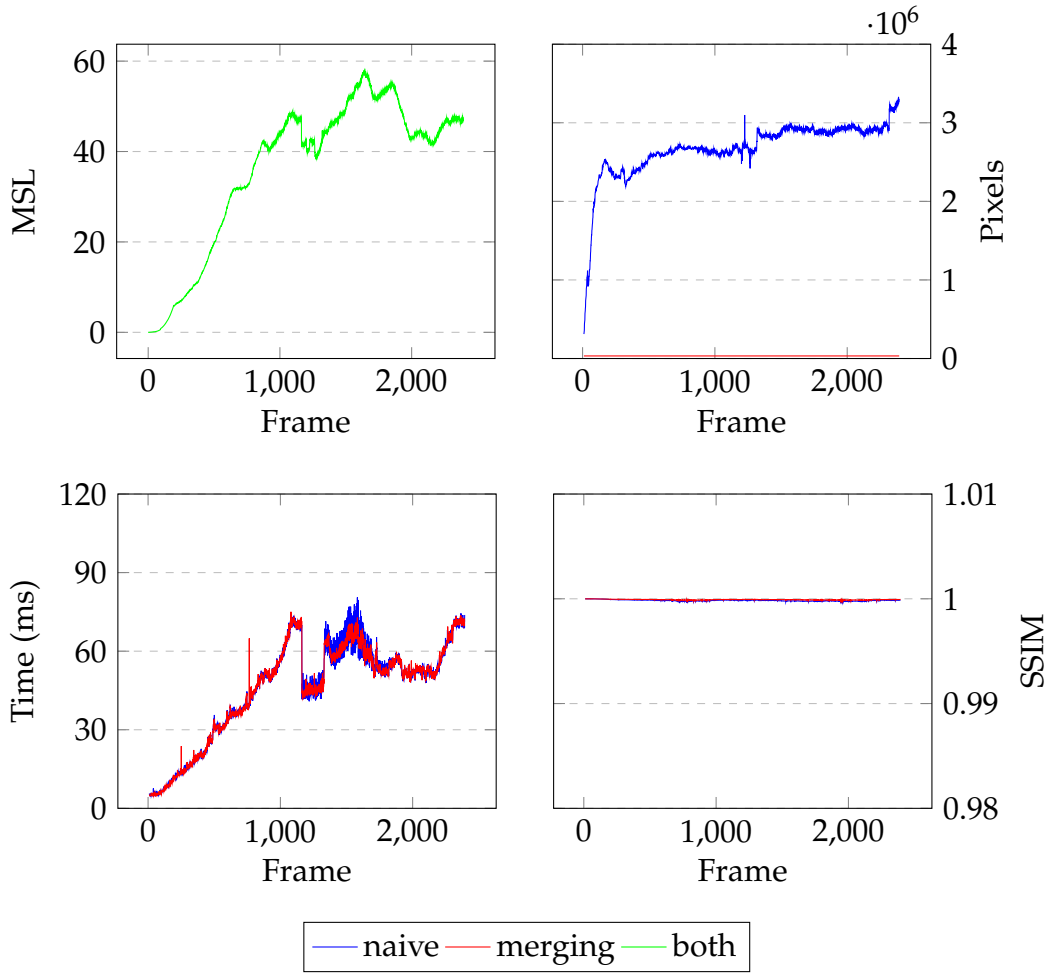


Figure 6.5: This test scene consists of multiple systems that are all different to each other.

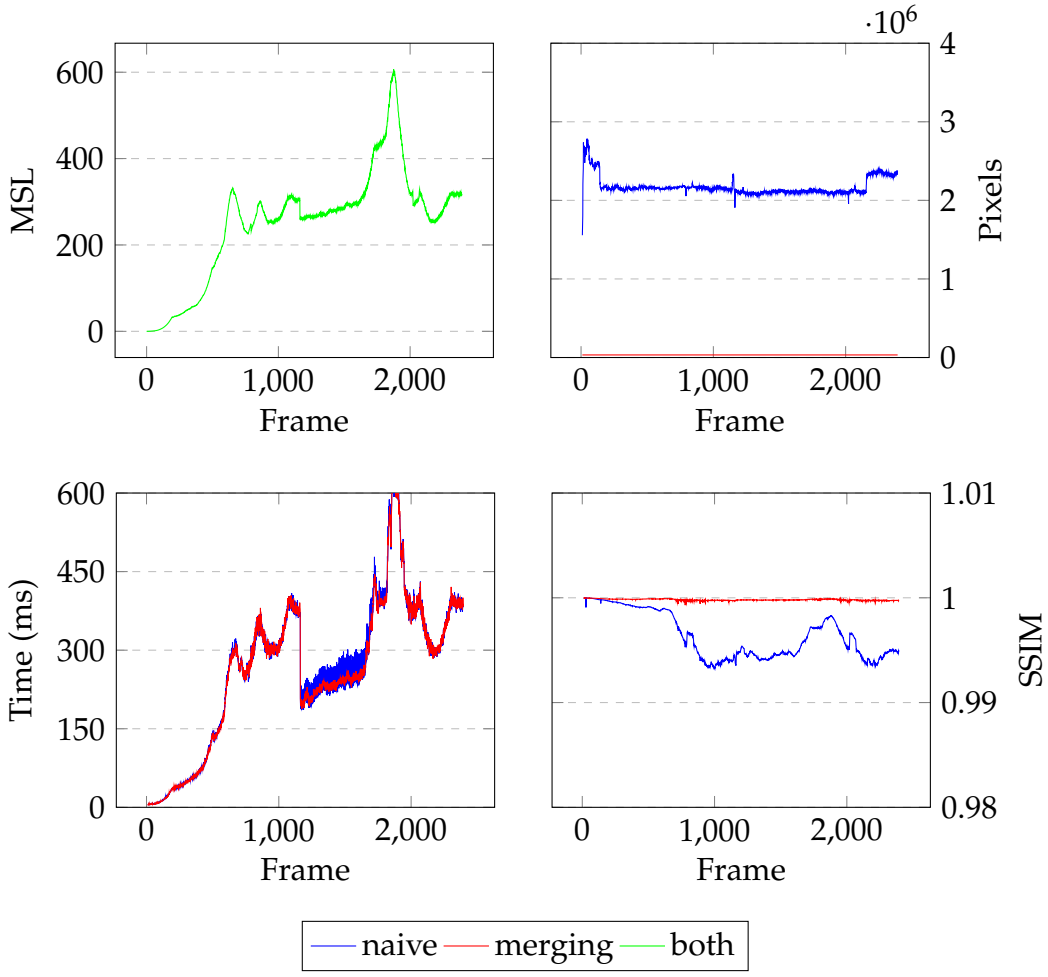


Figure 6.6: This test scene consists of several instances of multiple systems that are all different to each other.

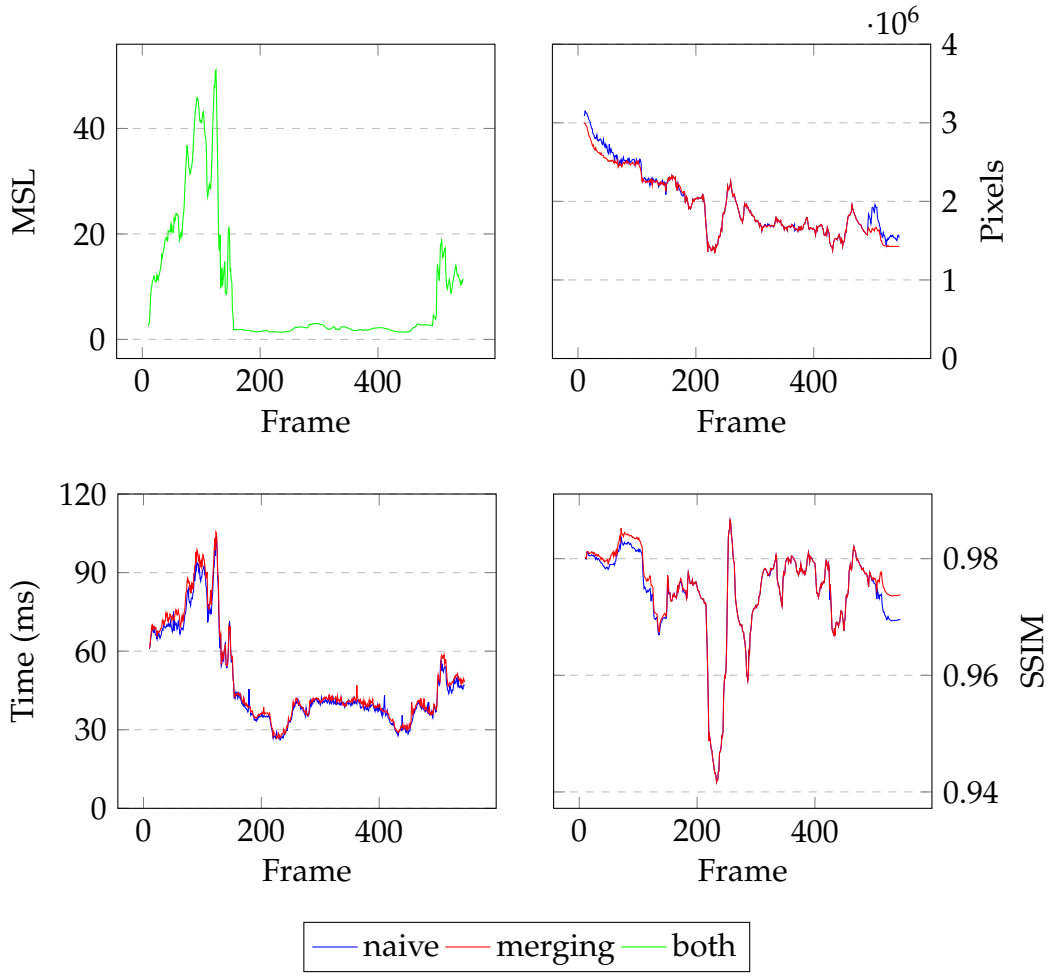


Figure 6.7: This test scene consists of robot lab and two systems representing ventilation smoke.

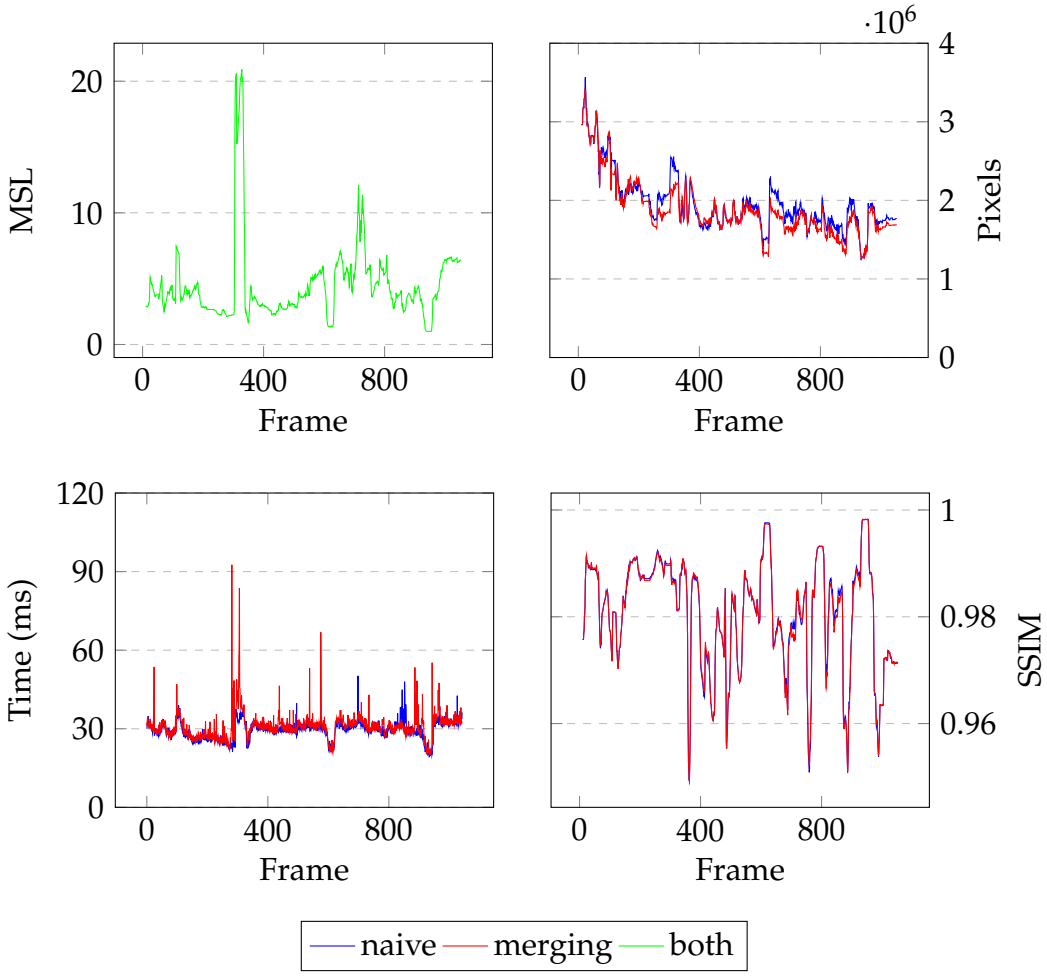


Figure 6.8: This test scene consists of viking village and several systems for fire effects.

It can be seen in figures 6.3 to 6.6 that the naive approach typically renders between two and three million pixels to the atlas per frame whereas the texture merging approach only renders between 10.000 and 30.000 pixels which is about 0.33%. As can be seen in figure 6.6 the SSIM is lower for the naive approach mainly due to the higher usage rate of the atlas. Since space in the atlas is more congested particles are rendered with lower quality to the atlas which also yields lower visual quality in the rendered image.

Although frame times are high in figure 6.4 and 6.6 this can be explained by the high MSL, which indicates that for a lot of pixels a lot of layers need to be sorted.

6.3 Particle Frame Times

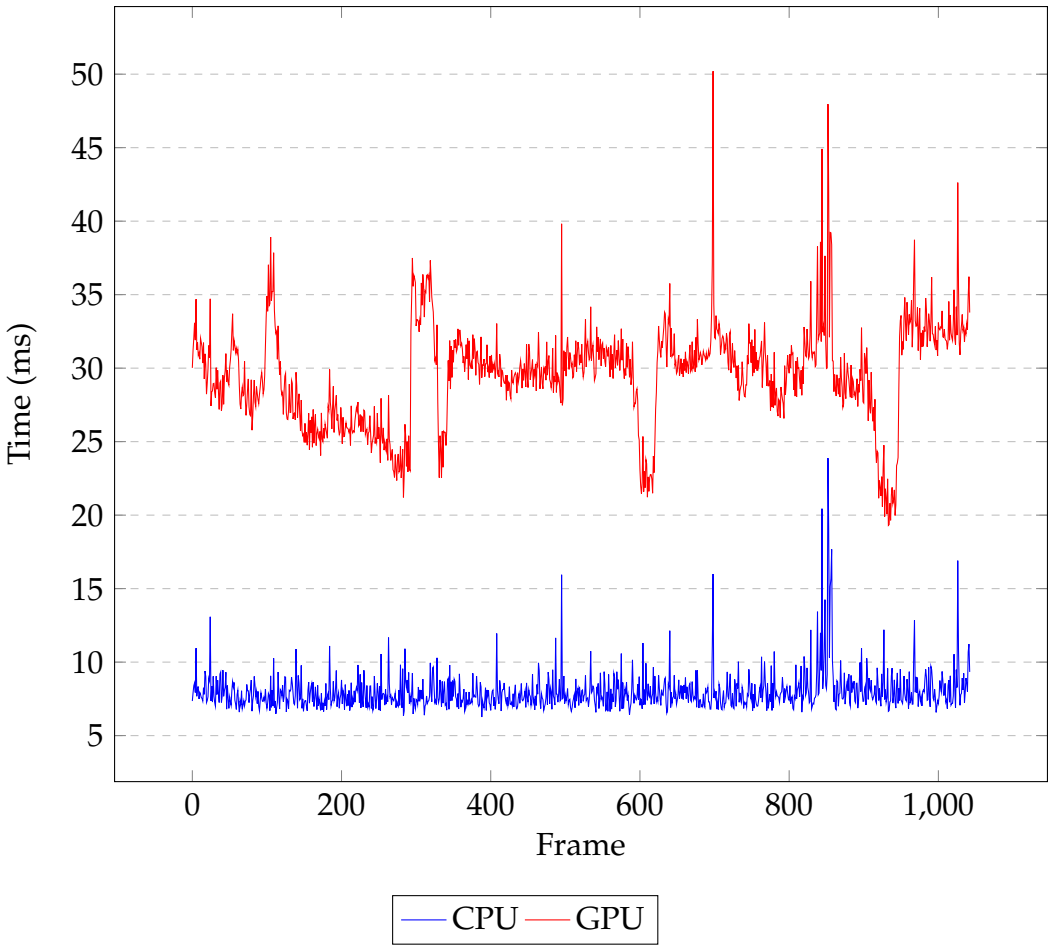


Figure 6.9: Frame Time Breakdown: Viking Village

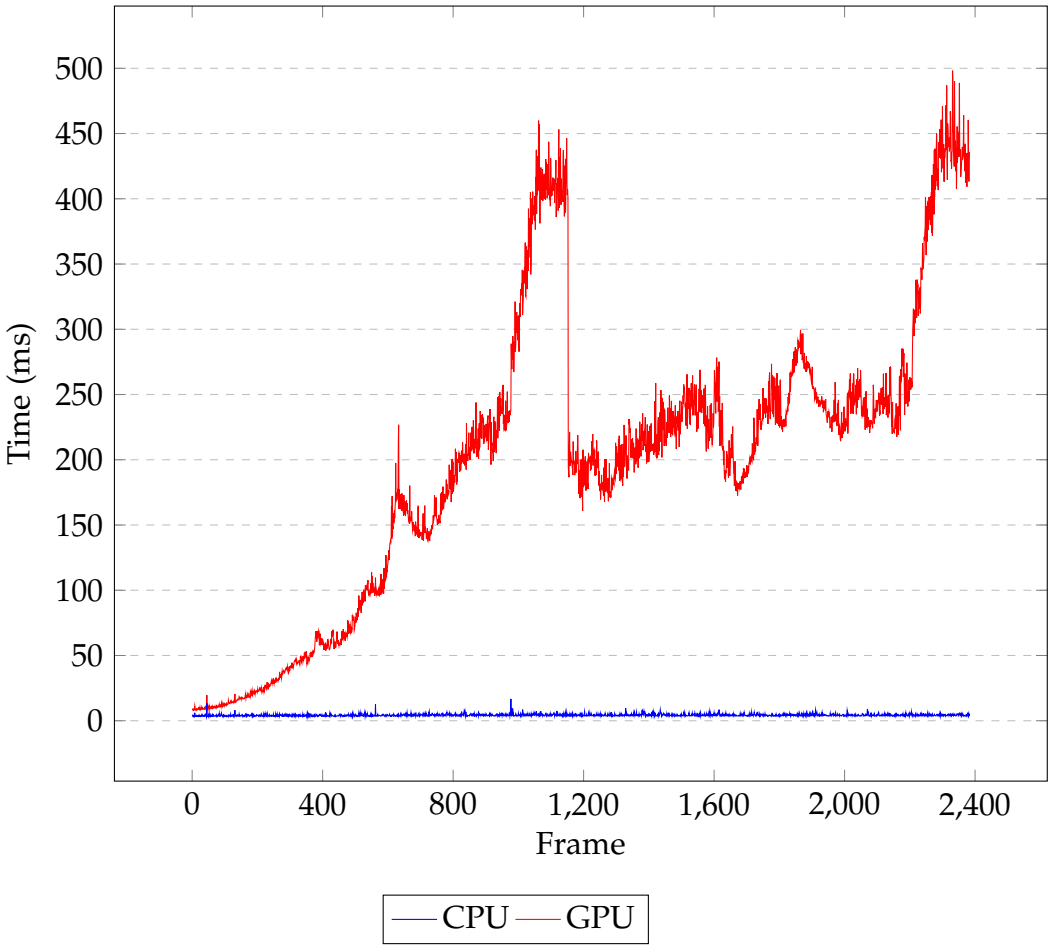


Figure 6.10: Frame Time Breakdown: Multiple Systems

It can be seen in figure 6.9 that CPU time for updating particles has an impact on performance since it contributes about 25% to the frame time. Still, it can be seen in figure 6.10 that updating the particles on the CPU is not a bottleneck for frame times but rather the rendering itself.

7 Discussion

State of the art IBR solutions for latency hiding like Time Warping [Mark, McMillan, and Bishop (1997)] suffer from major image quality issues during drastic changes of the head pose. The main problem are disocclusion artifacts which are even more visible when displaying transparent geometry. The proposed solution minimizes those effects by using an object-space based approach where not only currently visible geometry is considered but also a predicted set of possibly visible geometry. This means that disocclusion artifacts are only visible when the prediction is not quite right.

In order to provide a system that is wireless, the memory footprint of the MPEG stream, namely the atlas, is very important. As can be seen in chapter 6 the amount of additional pixels rendered to the atlas is typically in the range of 2–5%.

The main problem of the proposed solution is currently the rendering of complex particle systems. Although rendering time per frame is only increased by fifteen milliseconds for real life scenes on average, there are still some problems when there are a lot of particles on screen or when moving the camera into a particle system. The first problem is not only a problem to this implementation but also a problem for state of the art rendering engines. The second problem, however, happens due to the exact sorting of transparent fragments. There exist some approaches that may help to overcome this issue which will be explained in detail in the next chapter.

8 Outlook

As mentioned in the previous chapter there are currently some problems when rendering complex particle systems. Instead of using the exact sorting method of fragments there exist several approaches to approximate the sorting of particles.

Additive Blending

Instead of having the order dependent calculation for blending fragments together another approach is to use additive blending where fragments are added together. This results in order independent blending and therefore a huge performance gain by leaving out the sorting of fragments. This is preferably for particle effects like fire since adding the particles together gives the nice illusion of the fire emitting light. The problem with our framework is the lack of knowledge on the client how a fragment should be blended. This would require meta information to be sent to the client.

Bucket sorting

For particle systems it is often not required to sort the particles in perfect order but it should rather be accurate enough to have them bucket sorted. This approach defines a finite number of depth values where particles are then assigned to. This leads to a speed boost since sorting time is decreased.

Other sorting methods

Most modern engines provide sorting criterions for particle systems other than camera based. A nice approximation of camera sorting for some effects is for example sorting by age. With this approach there would be the same problem as with additive blending where we would need to know the age of a particle on the client in order to sort it correctly.

Adaptive Transparency

Since depth peeling removes the need to sort fragments on the GPU, it would be worthwhile to look into an adaptive approach of particle rendering that switches between linked list rendering and depth peeling whenever there are a lot of particles on the screen. The number of rendered layers dictates the loss of image quality.

Vertex data

No optimizations were made regarding the transmission of vertex data which could be a point of improvement in the future. Currently the position and rotation of particles are transmitted by using the standard vertex data. Since particles are always presented as billboards that may have a rotation and scale applied, instead of sending the vertex data redundantly for each billboard corner, the position, rotation and scale could be sent instead to save some bandwidth.

Appendix

Bibliography

Bavoil, Louis and Kevin Myers (Jan. 2008). *Order Independent Transparency with Dual Depth Peeling* (cit. on p. 21).

Blinn, James F. (July 1977). "Models of Light Reflection for Computer Synthesized Pictures." In: *SIGGRAPH Comput. Graph.* 11.2, pp. 192–198. ISSN: 0097-8930. DOI: 10.1145/965141.563893 (cit. on p. 18).

Everitt, Cass (2001). *Interactive Order-Independent Transparency* (cit. on pp. 9, 12).

HTC (2016). *HTC Vive*. URL: https://www.vive.com/media/filer_public/vive/product-overview/vive-hardware-hmd-1.png (visited on 07/19/2019) (cit. on p. 4).

Krishnamurthy, Venkat and Marc Levoy (1996). "Fitting Smooth Surfaces to Dense Polygon Meshes." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, pp. 313–324. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237270 (cit. on p. 6).

Lujan, M. et al. (Feb. 2019). "Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server." In: *2019 International Conference on Computing, Networking and Communications (ICNC)*, pp. 777–781. DOI: 10.1109/ICCNC.2019.8685588 (cit. on p. 17).

Mark, William R., Leonard McMillan, and Gary Bishop (1997). "Post-rendering 3D Warping." In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. I3D '97. Providence, Rhode Island, USA: ACM, 7–ff. ISBN: 0-89791-884-3. DOI: 10.1145/253284.253292. URL: http://doi.acm.org/10.1145/253284.253292 (cit. on pp. 15, 49).

Mueller, Joerg H. et al. (Nov. 2018). "Shading Atlas Streaming." In: *ACM Transactions on Graphics* 37.6. DOI: 10.1145/3272127.3275087 (cit. on pp. 1, 5, 9).

Murphy, Glenn (2019). *Explosion Sheet*. URL: http://glennmurphy.weebly.com/uploads/2/0/4/7/20470066/t_explosionsheet.jpg (visited on 07/19/2019) (cit. on p. 25).

Reeves, W. T. (Apr. 1983). "Particle Systems - a Technique for Modeling a Class of Fuzzy Objects." In: *ACM Trans. Graph.* 2.2, pp. 91–108. ISSN: 0730-0301. DOI: 10.1145/357318.357320 (cit. on pp. 8, 9, 13).

Samsung (2015). *Gear VR*. URL: https://images.samsung.com/is/image/samsung/at-gear-vr-r322-sm-r322nzwaato-000000002-r-perspective-white (visited on 07/19/2019) (cit. on p. 5).

Steed, A. and S. Julier (Mar. 2013). "Design and Implementation of an immersive Virtual Reality System based on a Smartphone Platform." In: *2013 IEEE Symposium on 3D User Interfaces (3DUI)*, pp. 43–46. DOI: 10.1109/3DUI.2013.6550195 (cit. on p. 4).

TooManyDemons (2016). *Hologram Console*. URL: https://sketchfab.com/3d-models/hologram-console-bfbbb481e98e4be38774b1d0204c192c (visited on 07/19/2019) (cit. on p. 7).

Unity (2015). *Viking Village*. URL: https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140 (visited on 07/19/2019) (cit. on p. 3).

Unity (2018). *Robot Lab*. URL: https://assetstore.unity.com/packages/essentials/tutorial-projects/robot-lab-unity-4x-7006 (visited on 07/19/2019) (cit. on p. 35).

Vries, Joey de (2019). *Tangent Space Vectors*. URL: https://learnopengl.com/Advanced-Lighting/Normal-Mapping (visited on 07/19/2019). https://twitter.com/JoeyDeVriez. (Cit. on pp. 18, 19).

Yang, Jason C. et al. (2010). "Real-Time Concurrent Linked List Construction on the GPU." In: *Computer Graphics Forum* 29.4, pp. 1297–1304. DOI: 10.1111/j.1467-8659.2010.01725.x (cit. on pp. 7, 9, 12).

Zhou Wang et al. (Apr. 2004). "Image Quality Assessment: From Error Visibility to Structural Similarity." In: *IEEE Transactions on Image Processing* 13.4, pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861 (cit. on p. 35).