computer graphics & vision

Inffeldgasse 16/2, A-8010 Graz

www.icg.tu-graz.ac.at

TUG

“Stick and Stones”

An Augmented Reality Game Based on Sketch Detection and Interaction

Denis Poric

Inst. for Computer Graphics and Vision

Graz University of Technology, Austria

Master Thesis

09 September 2011

Supervisor:

Dieter Schmalstieg

Contact: Denis Poric denis.poric@student.tugraz.at

Abstract

This thesis presents a game, “Stick and Stones”, whose design is centered on interaction through sketching. The game is based on a previously existing demo application, which supports drawing virtual sketches on a touch-screen, tracking real sketches on a whiteboard, and physical interactions between virtual and real sketches. We extended this demo application into a two-player game. Each player’s avatar is assigned a number of health points at the game start, and the goal of the game is to reduce the health of the opponent’s avatar to zero – this is achieved by hitting it with virtual stones, sketched by the players on their touch screen. The game uses multiple natural feature tracking targets, all tracked within a common coordinate system. We extended the sketch-extraction and collision-detection algorithms to support the new game requirements. Our results show that real and virtual sketching can be seamlessly merged into a consistent game experience.

Keywords: natural feature tracking, augmented reality, interaction, game, sketching, multiplayer.

Kurzfassung

In dieser Arbeit wird das Spiel „Stick and Stones“ präsentiert welches auf dem Prinzip der Interaktion durch Zeichnen basiert ist. Das Spiel ist basiert auf einer bestehenden Demoapplikation welche das Zeichnen von virtuellen Zeichnungen auf einem Touchscreen, Tracking von echten Zeichnungen auf einem Whiteboard und Interaktionen zwischen virtuellen und echten Zeichnungen unterstützt hat. Dem Avatar von jedem Spieler werden am Anfang des Spieles eine bestimmte Anzahl von Lebenspunkten zugewiesen und das Ziel des Spieles ist es die Lebenspunkte des Gegners auf null zu reduzieren. Dies wird durch das Treffen der Avatare mit von den Spielern gezeichneten virtuellen Steinen vollbracht. Das Spiel benutzt Natural Feature Tracking marker welche in einem gemeinsamen Koordinatensystem getrackt werden. Wir haben die Algorithmen für die Extraktion der Zeichnungen sowie die Kollisionsdetektion erweitert um die neuen Anforderungen des Spieles zu erfüllen. Unsere Resultate zeigen das echte und virtuelle Zeichnungen zusammengefügt werden können um ein kontinuierliches Spielerlebniss zu erzeugen.

Begriffe: natural feature tracking, Augmented Reality, Interaktion, Spiel, Zeichnen, Multiplayer.

Table of Contents

Table of Figures

1 Introduction

During the last few years, advancements in the hardware development of mobile devices like smartphones, netbooks etc. have led to the creation of a vast market for mobile applications. There are many different types of applications currently available such as music players, chat clients, games and even e-Banking applications. Of these different application types, games usually are the ones utilizing the more “exotic” features of a mobile device like the motion sensors, the camera or the GPS receiver.

Augmented Reality (AR) games are games which combine the real world with additional virtual information. These games use the camera for capturing a live-video feed and exploit this video feed in some manner for their gameplay.

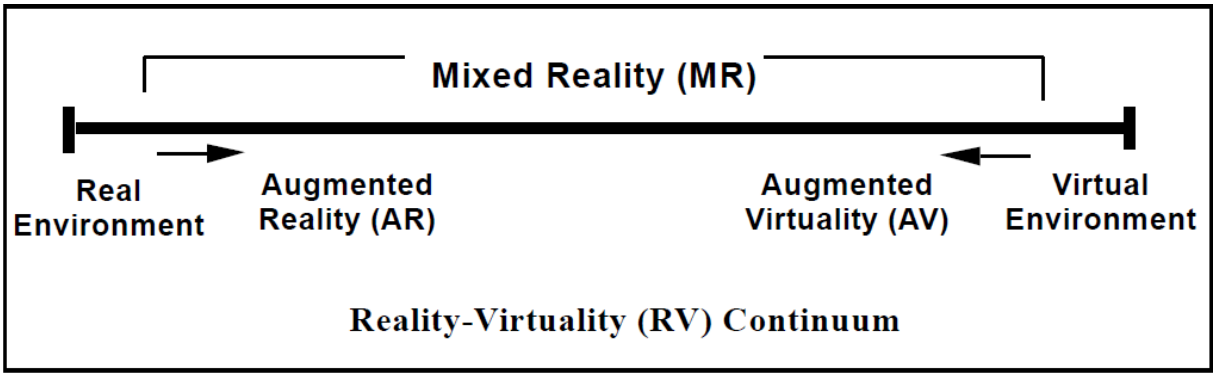

Figure 1: The Reality-Virtuality Continuum as defined in [13]

A formal definition of AR was given by Milgram et al in [13] as part of a Reality-Virtuality Continuum (see Figure 1). The two extremes of this continuum are the physical environment (i.e. anything that can be seen either in person or video) and the virtual environment (i.e. anything that is computer generated and invisible in person). Anything between these two points is called Mixed Reality. In this continuum AR lies closer to the reality extreme point and can be defined as "augmenting natural feedback to the operator with simulated cues". Another definition which tries to define AR without limiting it to certain technologies was given by Robert Azuma in [3]. He defines AR as a system that has 3 key properties:

- It combines real and virtual
- It is interactive in real time
- It is registered in 3D

The goal of this thesis was to design and implement a multiplayer game that seamlessly merges real and virtual sketches into a single, consistent AR game experience. We implemented the game on a Windows Mobile device using the Studierstube ES and Muddleware frameworks. Studierstube ES[1] is a general AR platform providing all the functionality needed for developing AR applications. Muddleware[2] is a framework intended for providing networking capabilities to AR applications which is already supported by Studierstube ES.

This thesis contributes to two distinct technologies that increase the robustness of the AR application and enhance the user interaction, as compared to previous approaches. Firstly, a multi-marker pose estimator is responsible for creating a global consistent coordinate system from multiple tracking targets. Each tracking target is a real-world object with a local coordinate system within the virtual environment, which is used for the placement of 3D objects within this environment. By using the multi-marker pose estimator, users switch between different tracking targets while the 3D objects within the virtual environment maintain their positions relative to the global coordinate system. Additionally, we use natural feature tracking (NFT) so that markers can be integrated into the real world background seamlessly. Secondly, we use sketch extraction and collision to detect if and in which direction an object within the virtual environment is allowed to move. This is accomplished by detecting real world obstructions once the object starts moving.

The multi-player requirements of the thesis led to a server-client architecture. The basic application functionality is provided by the Studierstube ES framework and its modules, the networking functionality is provided by the Muddleware framework. All remaining functionality is provided by custom created modules based on the above frameworks. These modules implement the game logic, sketch detection and extraction as well as the multi marker pose estimation and collision detection. The whole game consists of three entities: one server and two clients communicating over a wireless network. The two clients are synchronized through a database which contains gameplay information.

The thesis consists of five chapters. Chapter 2 is an examination of related work, particularly focusing on augmented reality games. Chapter 3 is about the game design, as well as the iterative process that led from the original design concept to

[1] http://handheldar.icg.tugraz.at/stbes.php
[2] http://handheldar.icg.tugraz.at/muddleware.php

the final design that was implemented. Chapter 4 describes in detail the technical implementation of the game. Chapter 5 presents the results achieved within this thesis work. Finally, Chapter 6 gives an outlook on future work.

2 Related Work

AR applications have a broad scope of use. Because of this, various types of AR application can be found on handheld devices as well as on stationary devices. Since this thesis deals with the design and implementation of an AR game, the related work will deal mainly with similar gaming applications either on handheld devices or PCs. The first 2 games presented here rely or sketching as a type of user interaction.

Figure 2: Sketchchaser game setup showcasing the VR objects drawn on a whiteboard [7]

The game described in [8] is called Sketchaser and has some similarities to the game developed in this thesis. It is a two-player game where the goal is to capture a flag with a virtual vehicle. Figure 2 shows the typical elements needed to be able to play.
Before the players can start playing the game, they need to draw the game environment where the game will take place. They do this by sketching visual elements on a whiteboard or on paper. These elements define either certain AR elements like trees, buildings, hills etc. or game concepts like player starting points and the goal. Example elements can be seen in Figure 3. The game recognizes these previously defined shapes and blends the appropriate AR elements in and out of the screen as these shapes are drawn and erased. The game allows the players to define the size of the AR elements. The AR elements are scaled up or down depending on the size of the shapes that the players draws. The players can control their virtual vehicles via mouse or joystick. The spatial tracking of the VR objects is

done via a shape recognition and pose estimation system called Nestor which is described in detail in [7].

Figure 3: Some example glyphs used in Sketchaser to signifly virtual objects. From left to right: Player1 and 2 starting positions and goal, grass and water patches, a building and tree, hills [7]

The similarity between this thesis and the abovementioned game is the high level of sketch-based user interactivity. Both games require the user to draw the game field where the gameplay will take place before the game can actually start. However in Sketchaser players can only draw in the real world. There is no way to add virtual objects without changing anything on the game background (whiteboard or paper), unlike the game developed for this thesis. In our game, players are able to draw own virtual objects (stones) and hurl them across the game field. Another difference is in the types of tracking that are used in both projects. Sketchaser uses a tracking system based on the analysis of the structure of contours of different shapes and their different concavities. According to [7] they reach approximately the same or better results when compared to tracking via fiducial markers or natural feature tracking (NFT). In contrast, we decided to use NFT for tracking and we seamlessly integrate markers into the game environment. The major difference between these two games is the way in which the multiplayer component is handled. Both games support two players. However, in Sketchaser both players are playing on one machine with two controllers; in the game developed for this thesis each player has a separate device and a server is used to synchronize them. The one device for one player condition was necessary because each player has to be able to move independently from the other player on the game field.

Figure 4: A sample gameplay sequence of Art of Defense showcasing what the environment looks like and what the players see [10]

The game described in [11] is called Art of Defense and is based on the "tower defense" game type. A tower defense game is a game type where the goal is to prevent a constantly increasing number of enemies from reaching the player's base, by placing "towers" which destroy these enemies along the route they take. Art of Defense supports two players playing on separate mobile devices connected via network. A typical setup can be seen in Figure 4. The goal of the game is to prevent as many enemies as possible from reaching the base of the players, which is located in the middle of the game field. This is accomplished by placing down defensive towers which destroy the incoming enemies. The game field consists of fifteen hexagonal tiles. At the beginning of the game only the base tile is present. By placing additional tiles, players reveal features of the map like terrain or enemy units. The players can position defensive towers on a tile's location by placing physical tokens on the tile; they can also upgrade them by placing differently colored tokens onto the tile. One interesting approach is to allow the players to draw the tokens before placing them on a tile, as can be seen in Figure 5. This sketch is processed by the game and the correct tower placed.

Both this game and our game are multiplayer games intended for a mobile device. Art of Defense also uses real world sketching as an interaction method. A major

difference is the tracking method that the games use. While the game presented in this thesis uses NFT, Art of Defense uses a more conventional approach (fiducial markers). The reasoning for not using NFT was a lack of computational resources on the mobile device. Similarly to Sketchaser, also Art of Defense allows only real-world sketches as a means to input game objects; in contrast to our game, no virtual sketches are present in the gameplay.

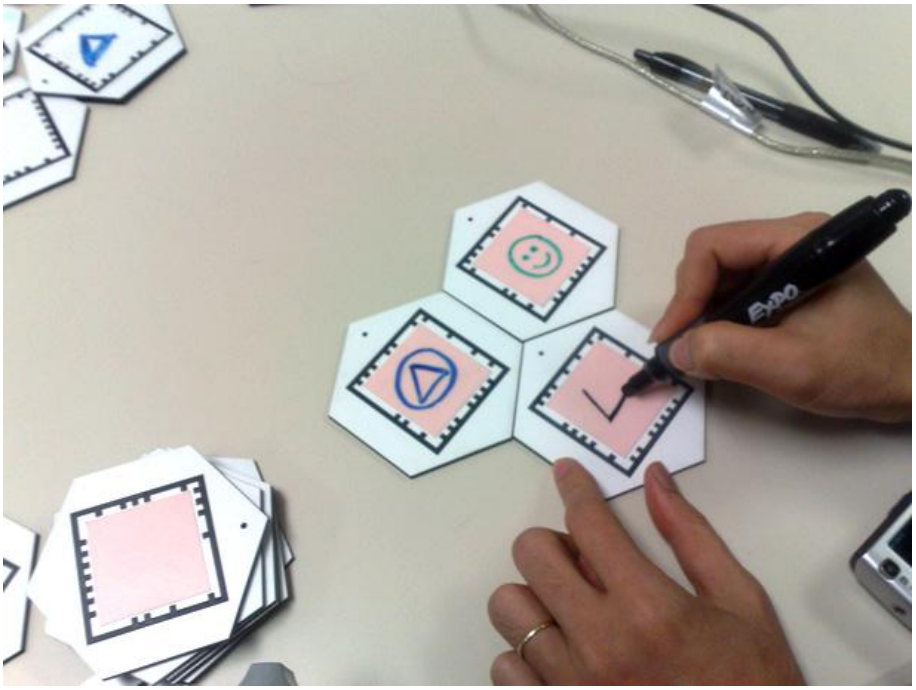

Figure 5: Art of Defense sketching system. Sketches define which kind of tower will be rendered. This approach was abandoned during development. [10]

The game described in [6] is called MoleARlert. This game is a multi-player variant of the popular Lemmings game created in 1991 by DMA Design. The goal of the game is to direct a certain amount of moles to the exit without them getting hurt on the different obstacles. MoleARlert is played on 12 x 12 cells outdoor area reminiscent of a chessboard. Players have two modes of directing the moles. The first is placing special markers, which direct the moles to a certain direction. The second is performing certain gestures. These gestures define actions which the first mole in the group can take, e.g. “build a bridge” or “dig through a hill”. To allow such a gameplay, this game consists of three independent modules. The first is an AR terminal which shows an overview of the game field (as presented in Figure 6); the second is the marker tracker which tracks the direction markers; the third is the person tracker,

which tracks the gestures of the players. These three systems are interconnected via a network.

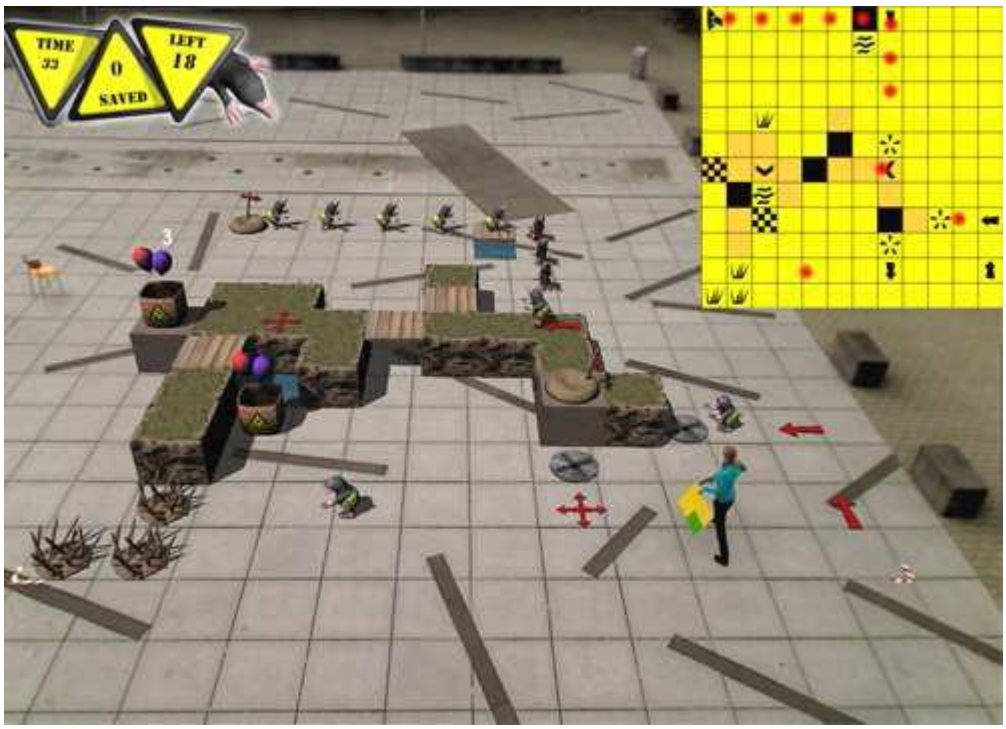

Figure 6: MoleARlert main view. Note the marker being carried by the person in the lower right of the image [5]

This game differs largely from the game presented in this thesis. First, the AR terminal is static and the camera is fixed. This does not allow the players to view the whole game field from a different perspective than a pre-defined one. The tracking systems are also different, and the player interaction is restricted to only the real world. This game does however show how diverse the different approaches can be, even with the relatively small subfield of games.

An example of AR game that allows an even larger degree of mobility is TimeWarp and is described in [10]. This game can be played alone or with multiple players. The goal is to find a mythological being – called Heinzelmännchen – within the city of Cologne, by searching for them in different time periods of the city. This is accomplished by using an AR display through which these beings can be seen at certain locations within the city. This display also allows players to explore certain city locations in different time periods. The game uses GPS tracking to display the virtual objects within the city. In order to play this game, users have to carry a personal digital assistant (PDA) which is used to display various information. In an early version of the game, players have also to wear goggles (as can be seen from Figure

7) on which some of the virtual content is visualized. In a latter version of TimeWarp, the whole game is handheld and based solely on an Ultra-Mobile PC (UMPC).

Figure 7: The setup used for TimeWarp [9]. The user is wearing goggles which display the VR objects and is using a PDA which is used to navigate around the city and solving the riddles

The major issue with TimeWarp is the tracking accuracy. GPS can be very unreliable within urban environments due to loss or weakening of the signal. This in turn causes instability in the visualization of the virtual objects – objects tend to drift or change locations randomly. Another issue with the early version of the game, is that although a PDA was used to display informations users had to additionally wear goggles which somewhat hampered them in enjoying the game.

Beside the above examples, there have been earlier examples of AR games developed for academic purposes. Treasures [4] and Pirates! [5] are not AR games but can be considered their predecessors. Treasures was developed for PDAs and allowed multiple players to play simultaneously. Virtual coins were placed within a certain area of the real world. The players had to collect these to earn points. Once a coin was found it had to be uploaded to a server which awarded the player a certain amount of points. In order to know when a player found a coin the game used GPS to track the locations of the players. Pirates! had a similar premise. Players started as novice captains of their own ships and had to complete missions to gain experience and gain a new rank. The game world consisted of islands which were virtual objects and which corresponded to a real world physical location. If players wanted to sail to another island, they had to physically move in the room. Since this game was

intended to be played indoors, it was not feasible to use GPS for tracking. Therefore a system of radio frequency proximity sensors was in used to locate the player.
One of the earliest true AR games was Invisible Train [25] (see Figure 8, left). The goal of the game is to prevent two virtual trains from colliding by operating various switches and junctions on the tracks. The game supports multiple players whose actions are synchronized via a network.

Figure 8: Examples of other academic AR games. Left Inivisible Train [23] right ARQuake[19]

Another early example of an AR game is ARQuake [20] (see Figure 8, right). This game is based on one of the first popular first-person shooter game Quake [12] developed by idSoftware. The goal of the game is to survive attacks from different enemies and reach the end of the level. The player of ARQuake wears goggles through which he or she can freely view the AR environment. The player can also interact with the virtual world with a physical device shaped as a haptic gun. The tracking of the player position in the virtual world is accomplished through GPS and fiducial markers spread around the real world. Another example is ARTennis [9] which allows two players to play a tennis game. The players use their phones as tennis racquets to hit the ball back to the opposing player. The game uses a simple physics system to simulate realistic ball movement and bouncing. The last example is more complex than the previous games. In Cows vs. Aliens [14] two teams of players try to save cows from aliens by leading them to safe areas where the team with most saved cows wins the game. The game field consists of pastures and two stables. These are represented by fiducial markers and each of these pastures can only have four cows at any time. The players need to select to which pasture the cows need to move by selecting a cow and sending it to an adjacent pasture. The game runs on a

server-client architecture. Each player carries a mobile device (the client). Any changes are transmitted via the server to all other players.

AR games are not only being developed for academic purposes. There are many commercial AR games available at the moment (see Figure 9). Games for mobile devices like the iPhone or the Nintendo DS usually use the inbuilt features of these devices like the camera, motion sensors and similar. AR games for gaming consoles usually require the console to have a peripheral device to collect data like the EyeToy for the Playstation 3 or Kinect for the Xbox360. Some currently available AR games for the above systems are: UFO on Tape [15] and Star Wars Arcade: Falcon Gunner [22] for the iPhone, A trading card game called Eye of Judgement [18] and a virtual pet game called EyePet [19] for the Playstation 3. There is also a similar game which is called Fantastic Pets [21] for the Xbox360.

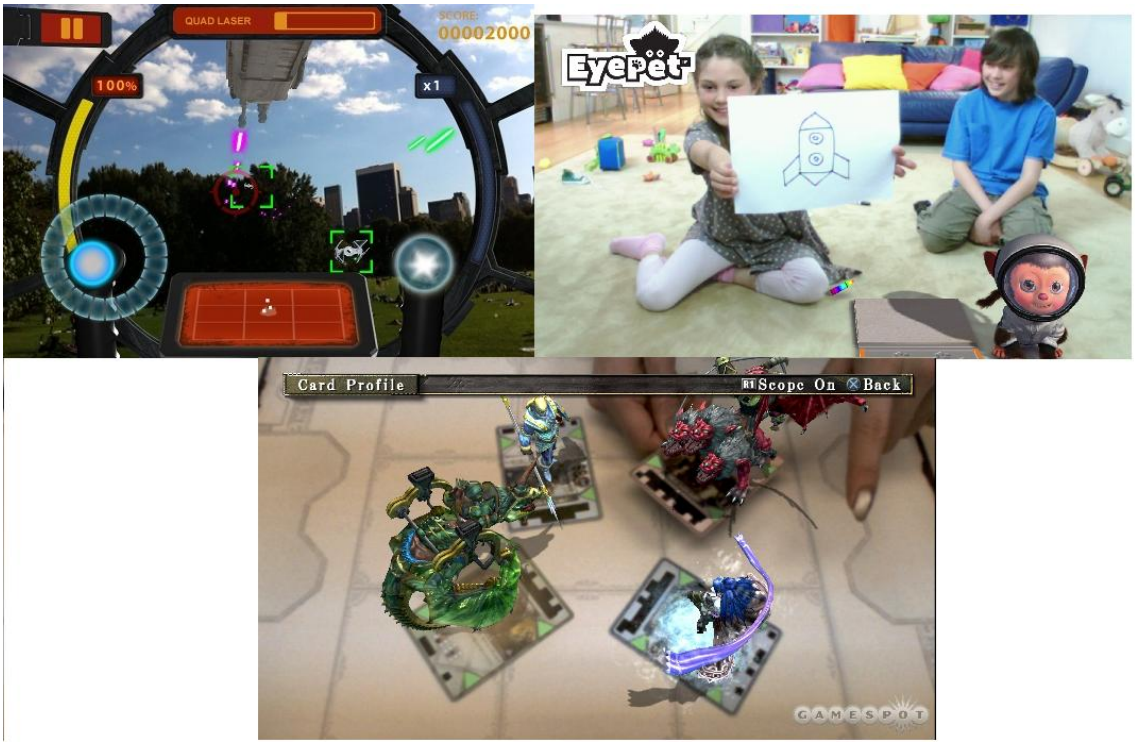

Figure 9: Examples of some already available commercial AR games. Upper left Star Wars Arcade: Falcon Gunner [20], Upper right and bottom EyePet [18] and Eye of Judgement [17].

3 Sticks and Stones game design

“Sticks and Stones” is similar to the Worms game [2]. Players physically draw the game field on a whiteboard with a black erasable marker, before the start of the game. The recommended shape the players should draw is a hill landscape or cave-like structure; however, the players may draw whatever shape they find interesting.

When the game starts, the players’ avatars are randomly placed on opposite sides of the game field. The players can draw virtual stones within a certain distance from their avatar, using the touch screen of their device; players select stones by clicking on them on the touch screen. Once a stone is selected it can be thrown in a desired direction. The players define the direction and the amount of force which will be applied on the stone using a rubber-band widget. This metaphor is similar to shooting with a slingshot: the longer one pulls away from the slingshot, the stronger the throw in the opposite direction. Similarly, the further away from the stone players pull the rubber band, the stronger the force that will be applied on the stone. The stone will be thrown along the direction opposite to the one of the rubber band. Each stone has a mass determined by its size. The stones are also affected by gravity. Therefore a stone with high mass will be thrown a shorter distance away then a stone with a smaller mass, when the same force is applied. The avatars can be moved around the field by clicking once on their figure and then clicking again on the desired destination. The avatars can only pick up stones within a defined distance, so they will need to move close to the stones to be able to use them.

The game is time-limited and turn-based. Each player has thirty-second turns to complete their tasks. If there is nothing a player can do he may choose to skip his turn by not executing any actions. At each turn switch, any stones that have not been used are removed from the game field.

Each avatar is assigned one hundred health points at the start of the game. If an avatar is hit by a stone, the avatar loses an amount of health points proportional to the mass of the stone. If the health pool of any of the avatars is reduced to zero, that avatar loses the current game round. Once this happens the game resets the avatar to random locations and starts another round. The game ends once one player has won two game rounds.

3.1 Design changes

The original game design iteratively evolved during the development of the game, as it became clear that certain aspects had to be changed as either impossible to implement due to framework constraints, detrimental to the gameplay or not needed altogether.

The first design change was to the game goal. The original goal was to simply kick the opponent's avatar out of the game field. This was changed to ensure a more dynamic gameplay. If the players were able to kick each other's avatar out of the game field, this could have shortened the game considerably. Due to the random spawning of the avatars it would be possible for one avatar to spawn close to the edge of the game field. If the other player started the game he would have an unfair advantage. Additionally, players with good aim could effectively end a game within a few seconds. Another issue that contributed to the change of the design was that the collision detection did not support any objects leaving the game field. In order to allow objects to leave the game field changes to the collision detection would have to be done which would have increased the memory consumption of the game, which was not feasible.

The second change was to the way the stones are placed in the game field. The original design had stones automatically spawn in two different locations. The first was directly next to the starting positions of the avatars where two to three stones would be deposited at the beginning of the game. Once these stones would have been used, additional stones would spawn in randomly chosen locations on the game field. The decision to change this aspect of the design had a number of reasons. The first was that it was impossible to foresee how the physical sketches that define the game field would be placed and if the stones would be reachable by the avatars, since the players draw arbitrary shapes on the whiteboard. This could lead to stones spawning in an inaccessible area and therefore not being usable by any avatar – if this happened, the game would be deadlocked. The second reason was to introduce a tactical element into the gameplay. If the stones were randomly spawned on the game field this could have caused players to play defensively by hiding avatars behind obstacles but still within the range of picking up the stones. This could make them inaccessible to the other avatar and would lead to a deadlock.

The third change was the removal of power-up items. With the change to the health pool system and the changes to the stone generation described above, the planned power ups (bouncy stone and jetpack) became a redundancy. Additionally, any power ups that would be spawned would have the same issues described above for the spawning of stones. Therefore, they were removed from the design.

The final change to the original design was the removal of a sudden-death round. This round was designed to make sure that the game would not draw on too long if players were stuck or ran out of projectiles. With giving the players the ability to create their own stones and increasing the mobility of the players this gameplay component became redundant.

4 Technical Implementation

The implementation of “Sticks and Stones” is based on a previous application (from hereon called *original application*), which was developed to showcase the interaction between virtual sketches and real sketches. Such application was implemented in C++ using the Studierstube ES framework and allowed the user to draw a virtual shape on a smartphone which then interacted with the real sketches on a whiteboard. “Sticks and Stones” is also implemented in C++ using the Studierstube ES framework, which provides most of the base functionality. This thesis will not explain in detail the Studierstube ES framework. More information can be found in [17].

The game can be divided into three large parts:

- Multi marker tracking
- Sketch extraction
- Game logic and support functions

The multi marker tracking part provides the functionality for tracking single and multiple targets. The sketch extraction part is in charge of creating a collision map by detecting sketches in the live-video feed; the map covers the whole game field and is used for collision detection. The game logic part handles all game-related functions like determining when a turn ends, which player is active, and what the score is. This chapter shows how the three parts of Sticks and Stones were implemented. Figures 10 and 11 show a more accurate composition of Sticks and Stones.

4.1 Multi Marker Tracking

The original application used fiducial markers for tracking and was able to track only one specific tracking target. In order to fulfill the requirements set by the game design the players need to be able to move their cameras on a large game area, to get a full overview of the whole game field, or to focus on a particular area of the game field. The issue with the original application is that it was very difficult to keep the marker constantly in sight while these different actions were going on. Constraining the game field to make sure that the marker is always visible is a very limiting factor for the gameplay.

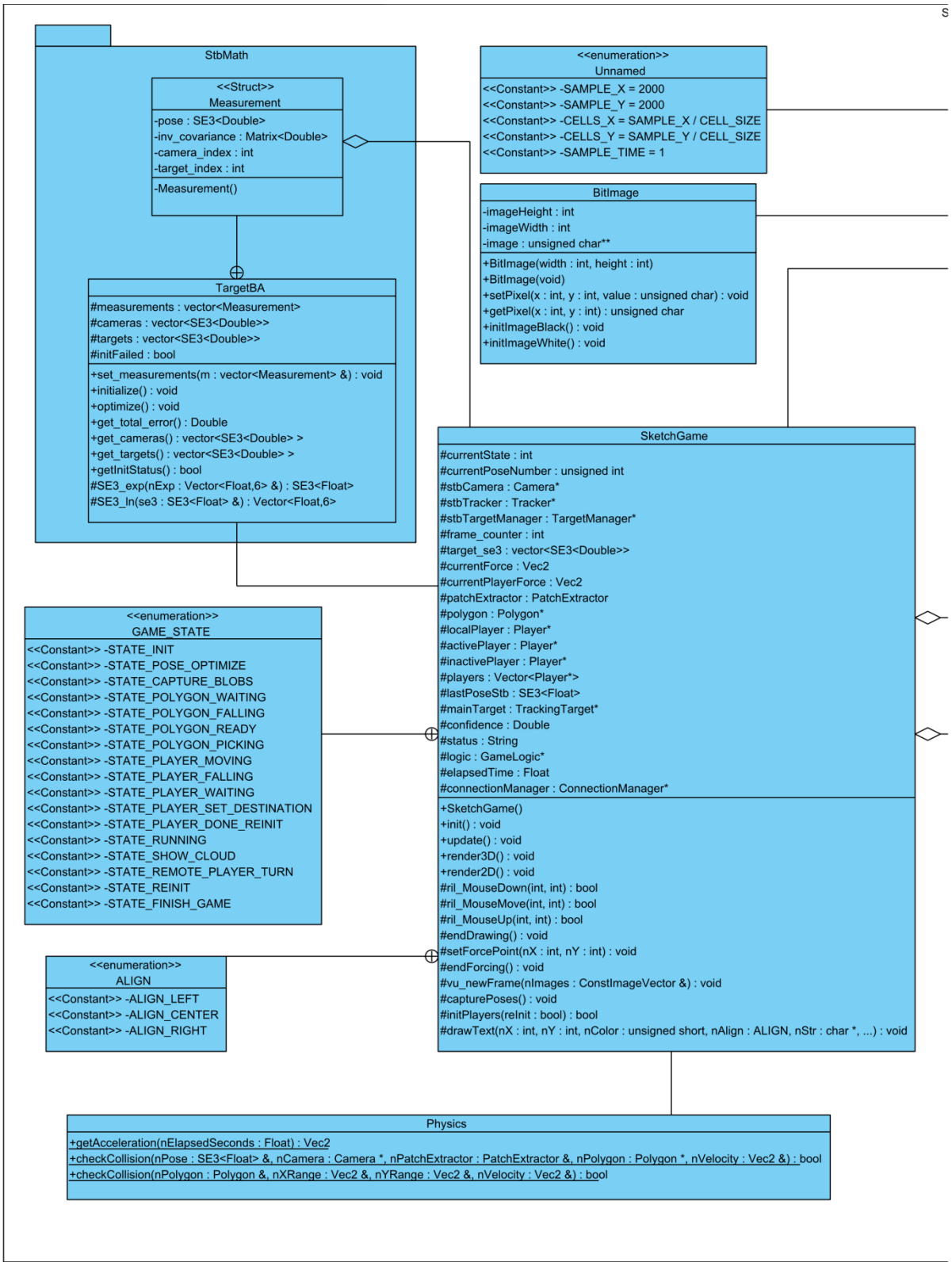

Figure 10: Part one of the class diagram of the application showing the main class SketchGame as well as the BitImage, Physics and TargetBA classes

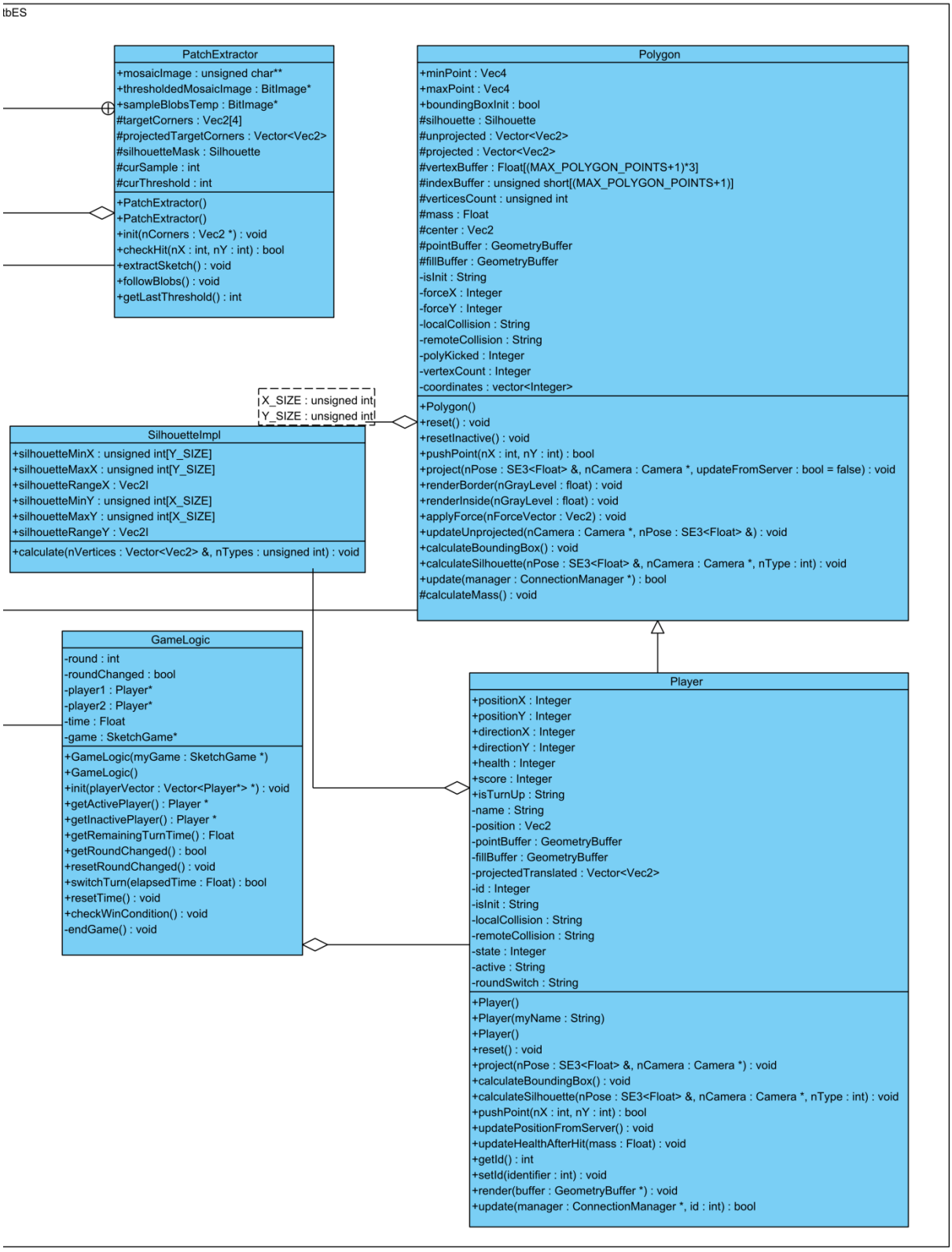

Figure 11: Part two of the class diagram of the application showing the PatchExtractor, SilhouetteImpl, GameLogic, Polygon and Player classes

In contrast, in this thesis we decided to allow the application to track multiple tracking targets (as seen in Figure 12), so that the players would have the freedom to move their cameras around a much larger game field. Furthermore, we switched to natural-feature tracking targets to increase robustness, as compared to fiducial markers. However, this solution presents an issue whenever a tracking-target switch occurs. Each tracking target defines a local coordinate system with the origin point on its center. This local coordinate system is used to estimate the pose of the camera within the virtual environment. Due to the fact that all local coordinate systems are distinct, the virtual objects (avatars and stones) take up different physical positions depending on which tracking target is currently in use; whenever a tracking-target switch happens, all virtual objects change their anchor on the screen and in the physical environment, since the local coordinate system has changed. The solution to this issue is to use a consistent global coordinate system, unique to all tracking targets.

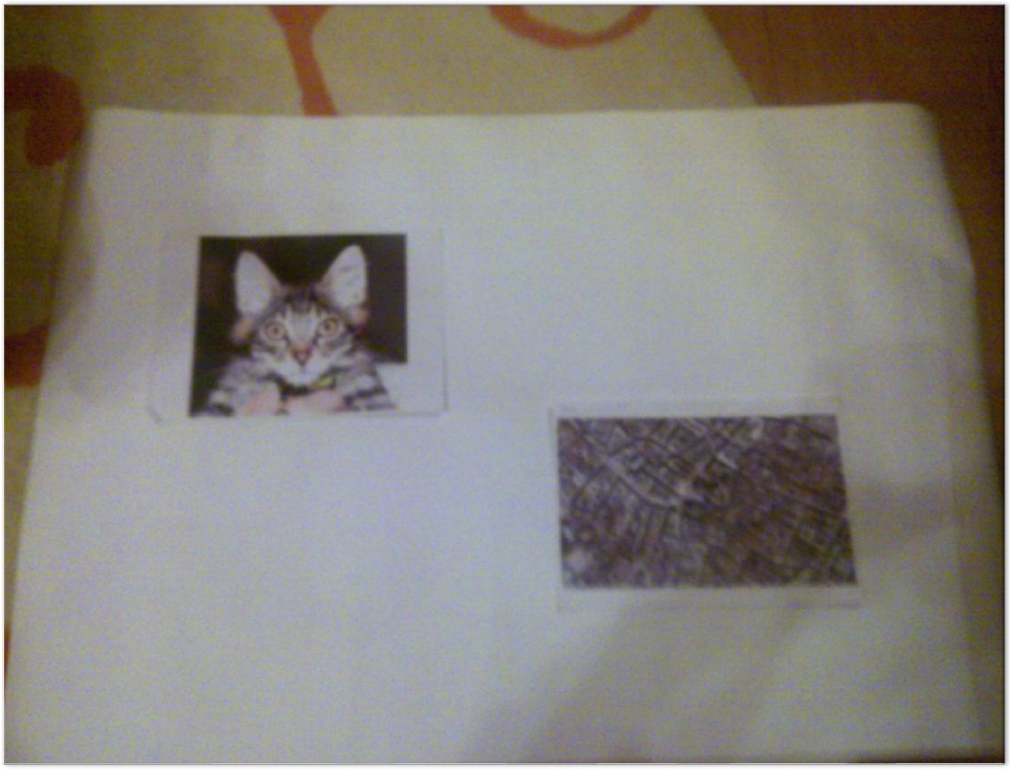

Figure 12: Tracking target general layout

This can be done by declaring one tracking target as the reference coordinate system, and converting the coordinates of all other systems into the coordinates of the reference system. We used a library written by Gerhard Reitmayr which was

originally implemented using TooN (Tom's Object-oriented numerics) [1]. This library inputs a number of measurements, each composed by the sequential camera-frame number in which the measurement was taken, the index of the visible tracking target and the pose matrix relative to it. After initialization the previously collected data is prepared and a sparse matrix (i.e. majority of values is zero) is created. The size of the matrix is defined by the following formula

$$F \times 6 + (T - 1) \times 6$$

where F is the amount of frames and T is the amount of targets. After this, a weighted least-squares Cholesky decomposition of said matrix is conducted. Once the decomposition has been completed, a matrix is expanded for each target from a sub-vector of the μ vector of the decomposition. These matrices are transformation matrices which transform one pose from the local coordinate system of the tracking target into the reference coordinate system. This enables the application to use multiple tracking targets as a unique, consistent coordinate system.

In this thesis, we implemented a number of changes to the original application. The first change was to switch to natural-feature tracking (NFT). While tracking via fiducial markers works well the markers themselves can be obstructive and break immersion. NFT targets fit better the game environment and are more robust, e.g. against partial occlusions. NFT is computationally intensive but at present feasible on phones [23].

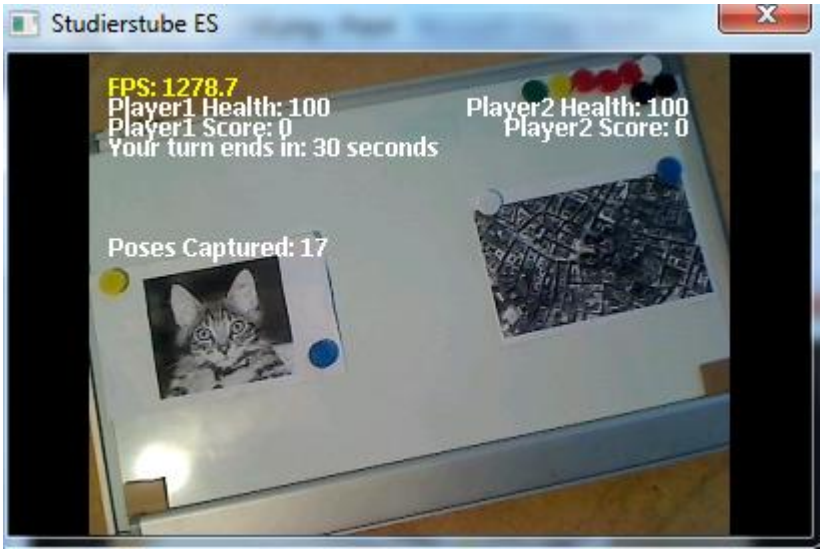

Figure 13: Typical view of the game field while capturing poses. The cat is the main tracking target

The second change was to integrate the pose-estimation library by Gerhard Reitmayr as a game component, porting it to the Studierstube Math library. The measurements which this pose estimator requires are stored in a vector by the game. Once the game has been started, each player is tasked to capture a certain amount of measurements. The capturing is started by clicking on the screen once, reaching the screen shown in Figure 13. At this point, players need to physically move the smartphone on the game field while keeping all tracking targets in the camera's field of view. Movement of the smartphone in a circular motion around the markers during the pose capturing has produced the best results during testing (see Figure 14).

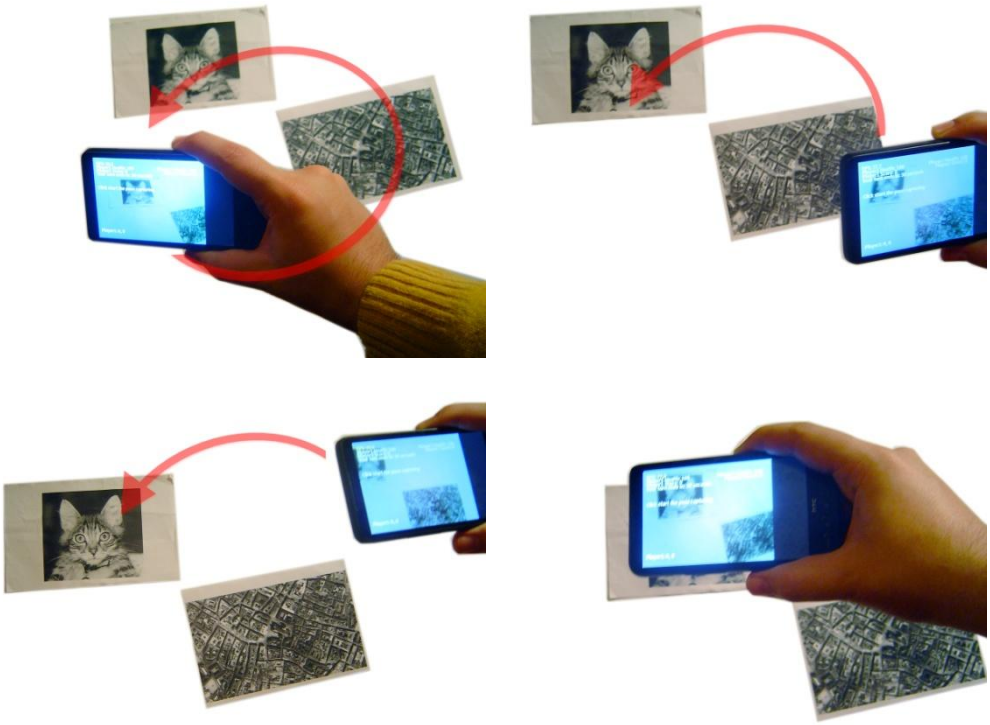

Figure 14: Sample movement of the device that results in best results for the pose estimator

The game captures the poses of all visible tracking targets automatically every ten frames. An earlier implementation required users to capture each pose individually. This was changed later, as it was a tedious solution to click the screen once for each measurement, especially as the number of required measurements increases. During testing, various numbers of measurements were used. Generally, the pose

estimation achieves higher accuracy when using more measurements. The game currently uses thirty-five measurements per tracking target, since testing has shown that this amount of measurements is a good compromise between accuracy and real-time performance. The pose estimation itself only produces a set of transformation matrices. The last step required to enable multi-target tracking was to decide a global coordinate system for all tracking targets. We achieve this by defining a main marker, whose coordinate system is considered as the reference one. While this marker is visible no transformations are executed. As soon as this marker is not detected anymore, the game goes through all visible targets and picks the first one it finds. Based on its index, the game then picks the appropriate transformation matrix.
The new pose is calculated according to the following formula

$$P_m = P_v \times M_v$$

where P_v is the pose of the currently visible tracking target, M_v is the appropriate transformation matrix which will transform the target coordinate system into the global coordinate system and P_m is the resulting pose which is finally used by the game. This whole process is done for every frame in which the main marker is not visible.
This approach is sensitive to tracking loss. During development two issues have been discovered. First, momentary tracking losses can disrupt the capturing of measurements. When this happens, the measurement vector is not fully filled due to the fact the game captures only the measurement for visible tracking targets. If a target is not visible, it will be skipped. During the initialization of the pose optimizer the internal arrays are filled with the data contained in the measurements vector. After this, the optimizer checks if all the values within these arrays are valid. If an invalid value is detected the initialization returns to the calling function immediately with a failure. To prevent this, the application checks the success of the initialization before starting the optimization. If the initialization failed, all measurements are erased and the user is prompted to restart the whole measurement capture process. A further safeguard has been implemented to detect if the amount of transformation matrices matches the amount of tracking targets. If this is not the case, the capturing of the measurements is restarted from the beginning. In general, for the pose estimation to succeed users must always focus the camera on all markers to make sure that the transformation matrices are all present and properly initialized.

4.2 Sketch Extraction

The original application extracted sketches on a per-frame basis. Each grayscale frame was fully processed, and a thresholding conducted to receive a binary image. The approach used was the adaptive thresholding described in [16]: the grayscale frame was divided into cells of predetermined size, and for each cell a mean threshold calculated. Pixel values where finally assigned in a binary image, depending if the pixel value in the grayscale frame was larger or smaller than the threshold. This dynamic approach is used to cope with local illumination differences. If a global thresholding approach was used, the segmentation would be wrong because these local illumination changes would not be taken into consideration. The binary image may contain gaps in the extracted sketches, usually caused by different thresholds used in adjacent cells. Therefore, the original application executed blob detection after thresholding. A new image was created and filled with white pixels. For each black pixel in the thresholded binary image, the neighborhood of the same pixel in the grayscale frame was examined. This neighborhood consists of the surrounding eight pixels, and for each of these pixels a gradient is calculated. The gradient was set to 255 (white) if the current neighborhood pixel in the binary thresholded image is zero. Otherwise, the gradient was set to the difference between the current neighborhood pixel and the central pixel of the current frame. For image storage, the original application used static two-dimensional unsigned-char arrays.
In this thesis, we implemented a number of changes to the original application. The first change is that the input for the sketch extraction was changed from the camera-frame size to a 2000×2000 image which covers the whole game field and is centered on the main marker. The size of this image was empirically chosen to cover the area of a commercially available whiteboard (60cm wide and 40cm high). Players must create this image before a game can start: each player is required to captures a certain amount of screenshots of the visible environment. Each screenshot is taken by clicking on the screen of the smartphone. Screenshots can be taken without all tracking targets visible, since at this stage the pose estimation is able to transform between the poses of the different tracking targets. While taking the screenshots, the player must be careful not to move their smartphone too far away from the whiteboard. While this will still capture the sketches and quickly cover the area needed to start playing, it will also degrade the quality of the detected sketches.

Therefore the players should try to take the screenshots while within a 20cm range from the markers. Additionally, extreme angles are also not recommended. Optimally the players should take their screenshots while viewing the scene perpendicularly. Each of these screenshots needs to be projected into the main marker coordinate system, since the screenshots are taken in camera or screen space. Due to the projection the screenshot can however be stretched and sheared which can cause gaps to form between the projected pixels. To prevent this, only the corners of the screenshot are projected into marker space and for each pixel which falls in the area between these corners an inverse bilinear interpolation is conducted. The resulting mosaic image is similar to the one presented in Figure 15. In order to know which pixels have already been set a binary image of the same size is used and each pixel that has been interpolated is set to *true* in it. After the interpolation is completed the amount of captured pixels is summed up and divided by the total amount of pixels. Once three quarters of the mosaic image have been filled, the game starts the sketch extraction. This value was chosen since it is a good compromise between precision and time needed to capture the game field.

Figure 15: Mosaic image created from 3 screenshots. They cover approximately three quarters of the 2000 × 2000 pixels game field

We also implemented changes to the sketch extraction part of the original application. Since the players are required by the game design to draw the game field before they can start the game, and this field stays static for the remainder of the game, there was no need to extract the sketches for every frame. Furthermore, since the input had changed to an image which encompasses the whole game field, the extraction per frame was not feasible anymore. Therefore the application was modified to execute the sketch extraction only once, after the three-quarter threshold has been reached. Also due to the change of the dimensions of the input image for the sketch extraction, all the internal image sizes of the sketch extractor had to be modified. Additionally, the cell sizes which are used during the adaptive thresholding had also to be changed due to the fact that the existing cell sizes produced worse results on the smartphone. Currently the game uses a cell size of twenty-five pixels for mobile devices and eight pixels for PC's. The above values were chosen

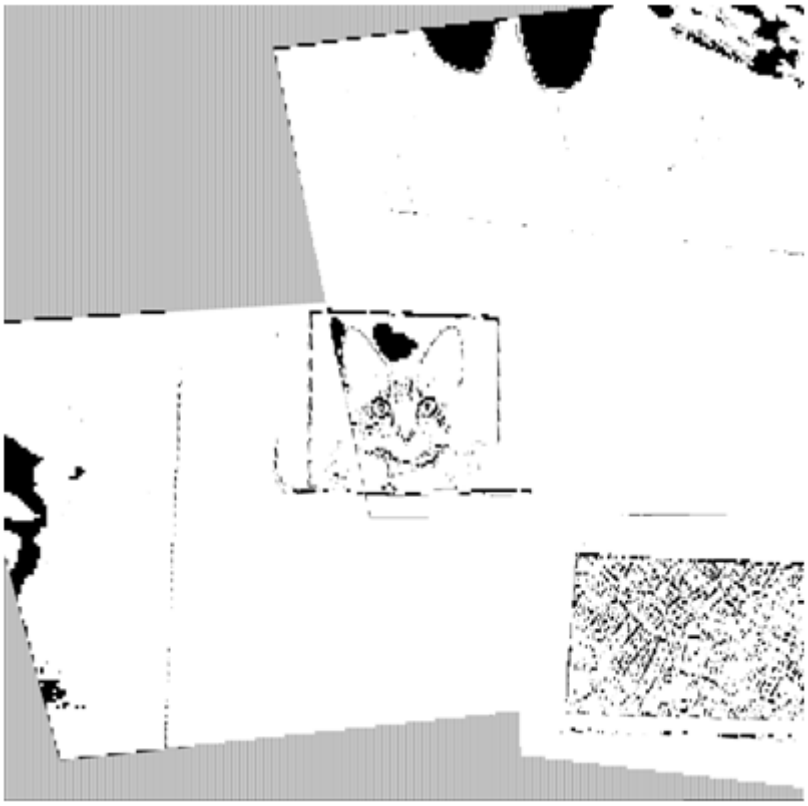

Figure 16: Result of the adaptive thresholding applied on the mosaic image presented in Figure 15

empirically, by testing with multiple values and choosing those where the resulting binary thresholding image for the same scene was the most accurate one and where the least false positives were identified. The results of the thresholding and the sketch extraction can be seen in Figures 16 and 17.

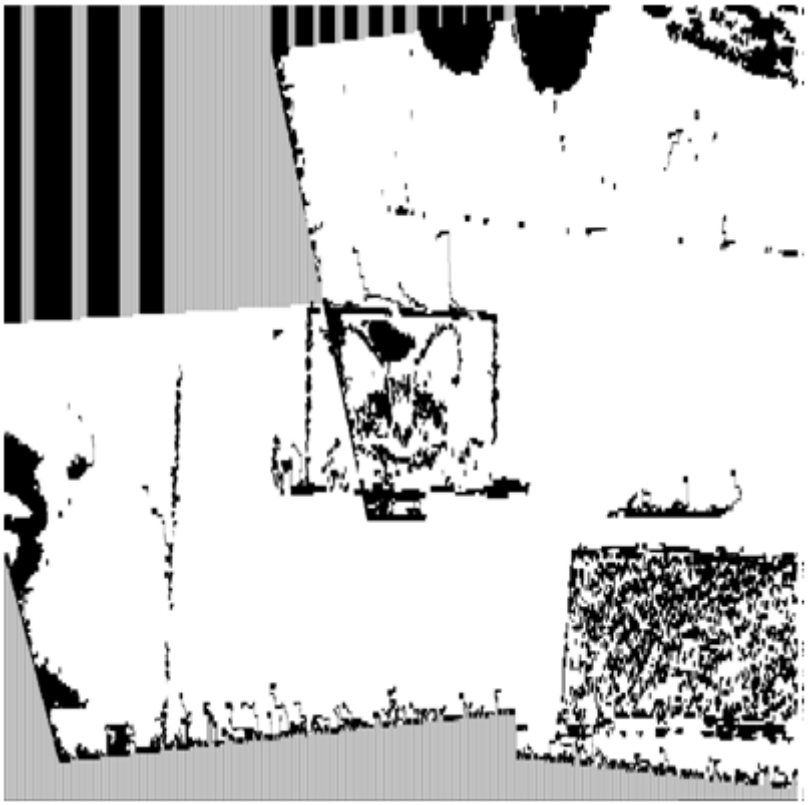

Figure 17: The result of the blob detection

Once the above changes were implemented, an issue surfaced when the game was being tested on a smartphone. As explained above, there are multiple images being manipulated simultaneously. The original application used unsigned char matrices to store the pixel data. Since the dimensions of the images were changed from 320 × 240 pixels to 2000 × 2000 pixels, it resulted in a massive increase in required memory. While this change did not cause any issues on the PC it proved to be too much for the smartphone due to the harsher memory restrictions for an application. On Windows Mobile 6 devices, all processes are given 32MB of memory for use. Since the game required a minimum of 12MB for three images – and additional memory for all the DLLs and the NFT target database – there was not enough memory available for additional allocations and the operating system would not allow any more memory to be allocated once the 32MB limit was reached. Exchanging the static arrays to dynamically allocated arrays resolved the issues, however when using different combinations of tracking targets a lack of memory still occurred. An examination of the images has shown that by using a whole byte (256 possible values) we were essentially wasting memory since the majority of images used in the

sketch extraction encoded a binary image where the pixels had only two possible values: zero for black and 255 for white.

We rectified this by implementing a bit-image class. The bit image itself serves as an interface to a two-dimensional character array where each byte stores eight pixels. The data is encoded by bit shifting and bit logic operations. To prevent code bloating, functions to access the array were implemented. These functions accept x and y pixel coordinates, ranging from 0 to 2000, and internally convert the x value to an appropriate range. This is done according to the following formula:

$$x_i = \left\lfloor \frac{x_b}{8} \right\rfloor$$

Where x_i is the input x coordinate and x_b is the resulting bit image x coordinate. While this give us the byte where the required bit is encoded the bit location is still required. To calculate the location of the required bit the following formula is used

$$i = x_i \; mod \; 8$$

Where x_i is again the input x coordinate, i is the location of the bit within the relevant byte and mod is the modulo operation. The introduction of the bit image class has reduced the binary image memory consumption by three quarters and has resolved all memory issues.

4.2.1 Collision Detection

The original application implemented a collision detection system. For each frame, in which a virtual object was moving or falling, the game checked if a collision happened. Once the collision check started, the application determined the dominant movement component by checking if the x- or the y-axis movement component was bigger. After this the silhouette of the virtual object was calculated as its extent within the virtual environment, stored as minimum and maximum coordinate pairs as well as per-pixel maximum x and y value for the x-axis and y-axis. After the silhouette was calculated, the application finally checked if the object was within the limits of the current frame. If this is not the case a collision was reported and the object stopped moving. Further checks were also done: the binary thresholded image which contained the extracted real sketches (similar to the one presented in Figure 17) was checked for a collision with the virtual object, which was handled in the main rendering loop.

In this thesis, due to the changes to the sketch extraction system, we implemented a few changes to the collision detection code. The change of the dimensions of the binary thresholded image, which is used for the collision detection, required a change to the silhouette system. The maximum x and y range values of the silhouette were changed to 2000.

Besides the collisions between real sketches and virtual objects, already handled by the original application, we implemented support for collisions between two virtual objects. These collisions can only happen between a polygon (stone) and an avatar object of two opposing players. We based the collision detection for this event on a bounding box intersection. For each frame the bounding box of the polygon and the opposing avatar are checked for intersections. As soon as an intersection is detected the opposing avatar is set in motion along the motion vector of the polygon. This type of collision also causes the avatar, which was hit, to lose health.

4.3 Game Logic

The final component of the game is the game logic. The game logic module is in charge of the gameplay-related actions, such as determining the score or controlling the player movement, handling collisions and similar tasks.

The original application was a technology demo and did not have a game logic module. The actions which the application was able to execute were controlled by a set of five states. The users were able to draw polygons on the screen by using the stylus and holding it pressed on the touch screen until they had drawn the desired polygon shape. The polygon itself was implemented as a separate Polygon class. It contained two buffers which were used to render the object; of these, one contained the polygon’s contour while the second was responsible for rendering the interior of the polygon. Besides these two buffers, there were also two vectors storing the coordinates of the vertices in screen space and marker space. When a polygon was drawn, vertices in screen space coordinates were added to the appropriate vector. Once the user lifted the stylus, no more vertices would be added and a projection of the coordinates into marker space would take place. After this, the polygon borders and interior would be rendered and the polygon would start falling along the direction the gravity was affecting it. If the user wished to move the polygon he or she clicked on the screen. While a polygon was present on screen this would cause the creation of a force vector which would be applied to the polygon on release of the stylus. This

force was also visualized by drawing a line between the polygon center and the current stylus location. The further the stylus was from the polygon at the time of release, the stronger the force applied. The application differentiated two polygon movements: falling and being thrown. Depending on which kind of movement was happening when a collision occurred, the polygon would behave differently. If the polygon was just falling after being drawn then it would just be stopped where the collision occurred. If it was however being thrown at that moment then the polygon would be destroyed and a cloud texture would be rendered to signify the crash.

In this thesis, we modified the original application to support the game logic. The first addition was the addition of a Player class. This was necessary because the game design needs virtual avatars or figures with which players can interact. The Player class was implemented by deriving it from the Polygon class. This was done as both objects are subject to the same collision and rendering routines.

The extreme points of a Polygon are calculated by looping through the screen-space and marker-space arrays and finding the minimum and maximum x and y values per array. These coordinates are then stored in a 4-element vector. For a Polygon, the first two coordinates of this vector are the x and y values in screen space while the last 2 are the x and y marker space coordinates. For a Player, only the marker space coordinates are saved. One of the requirements of the game design was the ability to hit players with stones, and this requires collisions between two virtual objects. We detect these by checking if a bounding box intersection happens between the objects. An intersection is detected by checking if the interval created by the extremes of one object intersect the extremes of the other object.

During testing it became apparent that some collisions would not be registered properly on both game clients. The issue was caused by the fact that only the screen location of the player was shared between the two clients. This caused projection and tracking errors to affect the collision detection. Additionally avatars would become distorted by movement across the game field. This was again caused by a short loss of tracking which caused the projection of the screen space coordinates to produce erroneous results. To resolve these issues we decided to switch the rendering and bounding box calculations of the player object to marker-space coordinates. Therefore, all functions which deal with filling the vertex buffer and silhouette calculations in the Polygon class were overridden in the Player class. Once this was completed, another issue presented itself in the silhouette calculation. The issue was

caused again by the projection of the coordinates. Since the main marker is located in the middle of the coordinate system, the projected coordinates could take on negative values. The silhouette calculation was not designed to work with negative values. Therefore the silhouette is calculated with a temporary array in which all marker space coordinates are shifted by 999 to make sure that the minimum coordinate is (0, 0).

The final change was the addition of the GameLogic class. This class is intended to supervise when a turn switch should occur and to report which player is currently active. Additionally, it is in charge of creating the Player objects during the game initialization. For every frame that is processed, the game subtracts the time required for the frame from the remaining time in the GameLogic object. If the remaining time is zero or lower, the game starts the turn switch. Due to the fact that every time the user interacts with the game the rendering pauses it may occur that the two game clients receive the command to switch turns at different times. This in turn causes the clients to lose synchronization and eventually leads to both clients waiting for the other to finish its turn. To prevent this, a state condition was added to the turn switching requirement as well. Both clients need to have the turn-switch flag set and they need to be in the STATE_PLAYER_WAITING state to execute a turn switch. The switch itself is relatively simple and is executed by switching the active player object with the inactive and vice versa. Additionally, the force vectors for the polygon and the direction vector of the players are set to zero and the remaining time of the GameLogic instance is reset to 30 seconds.

4.3.1 States

The original application was controlled by a state machine containing five states:

```
STATE_WAITING
STATE_POLYGON_FALLING
STATE_POLYGON_READY
STATE_RUNNING
STATE_SHOW_CLOUD
```

In this thesis, due to game design requirements, we introduced new states and repurposed some other states. New states regulate the avatars' movement, the application initialization, turn switching and player reinitialization. STATE_WAITING

was split into STATE_POLYGON_WAITING and STATE_PLAYER_WAITING. The new states are:

```
STATE_INIT
STATE_POSE_OPTIMIZE
STATE_CAPTURE_BLOBS
STATE_POLYGON_WAITING
STATE_POLYGON_PICKING
STATE_PLAYER_MOVING
STATE_PLAYER_FALLING
STATE_PLAYER_WAITING
STATE_PLAYER_SET_DESTINATION
STATE_PLAYER_DONE_REINIT
STATE_REMOTE_PLAYER_TURN
STATE_REINIT
STATE_FINISH_GAME
```

As can be seen from Figure 18, the game starts in the STATE_INIT state. This happens after the player and polygon objects are created and all necessary data is loaded. Once the user clicks on the screen, the game switches to the STATE_POSE_OPTIMIZE state. When in this state, the game collects poses for the measurements required by the pose estimator (as described in Chapter 4.1). Once enough poses have been collected and the pose estimator finishes his estimation, the game switches to the STATE_CAPTURE_BLOBS state. During this state, the game waits for the user to take screenshots of the game field. The screenshots are processed as described in Chapter 4.2. Once this is completed, the game switches to two different states, depending on which player is the active player. This is determined by which player connected to the server first. This player is marked as the active player while the other player is marked as the inactive player. The active player switches to the STATE_PLAYER_WAITING state while the inactive player switches to the STATE_REMOTE_PLAYER_TURN state. Since it is very likely that both players will not complete the pose estimation and sketch extraction at exactly the same time, a busy loop mechanism makes sure that the faster player waits until the slower player is done with the previously mentioned actions. From this point on, the active player state is stored in the shared database and used by the inactive

player to determine its current state. From the STATE_PLAYER_WAITING state the active player can enter three different states:

- STATE_SET_DESTINATION – the game enters this state if the active player has clicked on his avatar. If the player clicks on the screen again while being in this state, the game sets a destination point for the player object and switches to the STATE_PLAYER_MOVING state, which causes the avatar to move. The movement of the player is based on an interpolation of the player position between the origin and destination points.

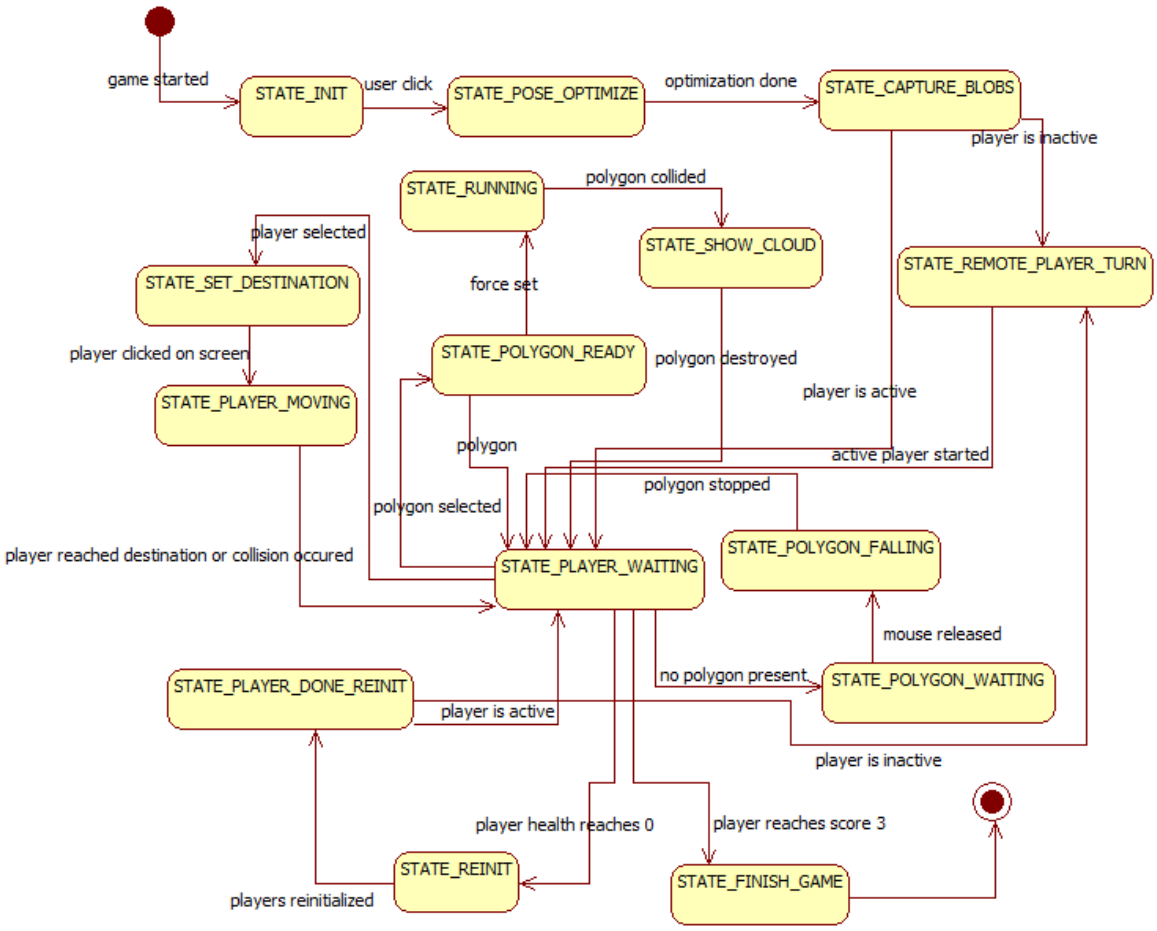

Figure 18: State machine of Sticks and Stones

The player object traverses this distance within five seconds. To know by how much the player needs to be moved in a frame the game uses the following formula

$$I = s \times t$$

where *s* is one fifth of the total distance between the original location and destination, *t* is the elapsed time since the movement started and l the

resulting position. Once the player is twenty distance units away from the destination point, a collision occurs or the player reaches the edge of the game field, the player figure is stopped and the game switches to the STATE_PLAYER_WAITING state again.

- STATE_POLYGON_READY – the game enters this state if the active player has clicked on a polygon object. If the screen is clicked again the game will start capturing the force points and will draw a line connecting the touch point with the center of the polygon. Once the touch is released, the state switches to the STATE_RUNNING state and the difference of the coordinates of the last touch location and the polygon centre is set as a current force which is then applied to the polygon. If the polygon collides with something or reaches the edge of the game field, the game switches to STATE_SHOW_CLOUD, which causes the destruction of the polygon. The polygon is replaced by a cloud texture which expands for a few seconds and then disappears. Once the cloud is gone, the game switches back to STATE_PLAYER_WAITING.
- STATE_POLYGON_WAITING – players enter this state if they click on any location of the screen but a player or polygon object. If the screen is clicked again the players will start adding vertices to the polygon object since the object is just reset and not deleted. If a polygon already exists on the screen then the game will not enter this state. Once the polygon has been drawn the game will switch into the STATE_POLYGON_FALLING state, which will cause the polygon to move along the defined gravity direction. Once a collision occurs or the polygon reaches the edge of the screen the polygon is stopped and the game switches to the STATE_PLAYER_WAITING state again.

The state changes of the inactive player work differently. They do not depend on any user input but on the states of the active player. The inactive player checks each frame what the current state of the active player is. Some states are ignored, like the STATE_POLYGON_READY, STATE_POLYGON_WAITING and all the initialization states (STATE_INIT, STATE_POSE_OPTIMIZE etc.), since these states are just

used to prompt the active player for some input, or states which cannot occur anymore. Rather, the inactive player checks if the states which should happen after these have been triggered and gets relevant data from the database (e.g. polygon vertex coordinates if the active player's state is STATE_POLYGON_FALLING).

The game design defines that the game is played in multiple rounds. This requires the ability to reinitialize the players and reset the game field and player locations. To accomplish this, the STATE_REINIT was introduced. If, during the collision between a polygon and avatar, the health of the avatar goes below zero then the reinitialization state is triggered. Once this state is registered by the inactive player (i.e. the player that was hit by the polygon) it is also triggered in the active player and both clients display a message urging the user to click on the screen. Once the user clicks, a similar busy loop is used as at the end of the STATE_CAPTURE_BLOBS state. Both clients reinitialize their respective avatars and update the database with the new values. The only value which is kept is the player score. After this, the players switch to STATE_PLAYER_DONE_REINIT and check the state of the other player in a busy loop. Once both players are in this state they exit the busy loop and the active and inactive players are set to the same states as at the beginning of a new game, STATE_PLAYER_WAITING and STATE_REMOTE_PLAYER_TURN respectively.

The last state that both players enter is STATE_FINISH_GAME. This state can be triggered by two actions:

- By clicking outside the area covered by the video feed
- By winning three rounds of the game. Once the game logic detects that the score of one of the players is three it sets the state of the game to the STATE_FINISH_GAME. When this happens a message is displayed to both players indicating who won and who lost. When a player clicks on their screen while this message is present, the game is exited.

4.3.2 Player Synchronization

One of the difficulties encountered in this thesis is that two players share the same game field. They have to be able to see what happens with the other player without having to look at his or her screen. Therefore, some method had to be introduced to synchronize the two clients. The method that we used is the Muddleware networking

solution, explained in [24]. The synchronization works by sharing certain variables of the Player and Polygon objects via a server-side application. This server administrates a database contained in a XML file which stores the shared elements in XML element form. The database structure is shown below.

```
<?xml version="1.0" standalone="no" ?>
<DB>
<Player />
<Player/>
<Polygon/>
</DB>
```

It contains three elements: two player and one polygon elements. All the shared variables are encoded as attributes of the appropriate XML elements. An exemplary player entry is shown below.

```
<Player id="1" PositionX="-549" PositionY="-380" DirectionX="316"
DirectionY="116" isInit="false" localCollision="false"
remoteCollision="false" state="0" active="true" health="100" score="0"
isTurnUp="false" roundSwitch="false" />
```

The *id* attribute defines the player, while *PositionX* and *PositionY* define the location of the center of the player. *DirectionX* and *DirectionY* are only set if the player is moving and indicate the destination. The attribute *isInit* is used in the initialization by the application to check which player has already been initialized. The *localCollision* and *remoteCollision* attributes are used to signal to the game clients where the collision occurred and to stop the player movement if one of these flags is set. The *state* attribute is used to share the current game state. *health* and *score* encode the current score and health values. The rest of the attributes (*active*, *isTurnUp* and *roundSwitch*) are used to indicate turn switches and round switches.
The Polygon element is similar in structure.

```
<Polygon isInit="true" localCollision="true" remoteCollision="false"
polyKicked="0" forceX="0" forceY="0" vertexCount="25" coordX1="-506"
coordY1="-226" … />
```

The *isInit* attribute indicates to the game if the polygon has been initialized, similar to the player. *localCollision* and *remoteCollision* have the same task as their Player counterparts, while *polyKicked* is used to differentiate if a polygon was kicked or is falling. The next two attributes, *forceX* and *forceY*, are set when a polygon is kicked, and store the x and y components of the force vector. The *vertexCount* attribute indicates how many vertices the polygon has. The last attributes are the marker space coordinates of the vertices. There are twice as many as the number of vertices, and they are numbered starting from 1 (i.e. coordX5/coordY5 is the fifth vertex-coordinate pair).

In order to share the attributes, the Polygon and Player classes are derived from the Muddleware interface Element. Each class has variables which match the above-mentioned attributes. These variables are updated by two methods. Variables for which it is not essential to immediately send the updated values to the database are updated before each rendering pass if their values have changed in the database. Any changes are uploaded to the database after the frame has been drawn. Any variables like the state of the active client, the flag indicating if a collision was detected or not and other similar ones which require immediate updating use a direct call to the Muddleware system, through the method sendAndReceive().

In order to connect to the database, the configuration file needs to be edited by adding the host to which the game should connect. Since the database is required to play the game, if the game cannot connect to it due to network issues or an error in the configuration file, it cancels the initialization of the rest of the application and ends the application.

5 Results

The goal of the thesis was to design and implement a game which allows seamless real- and virtual-sketch interaction within an AR game. In this thesis, we took an original application that demonstrated sketch-based interaction in AR, and extended it to a game called Sticks and Stones. We integrated multi-marker tracking and modified the sketch extraction and interaction systems to allow two players to play the game over a wireless network. The game was deployed and tested on a commercially available HTC HD2 mobile phone running Windows Mobile 6.5 Professional with a 1GHz Qualcomm Snapdragon CPU, AMD z430 GPU and 576MB RAM. The game plays fluidly with a frame rate of approximately 15 to 25 frames per second, depending on the scene that is being viewed. This frame rate can be improved by connecting the phones to their charging modules. To play the game, besides the elements presented in Figure 19, a separate wireless-enabled PC is required .

Figure 19: Elements required to play the game: NFT targets, whiteboard, mobile phone and black erasable marker

The game is started by starting the StbES executable. The first scene that the players see is a video feed from the camera and a message prompting them to click

once on the screen to start the pose capturing, as seen in Figure 20. If no markers are visible or tracking is lost, the game displays a red warning message in the center of the screen. Once the player clicks on the screen the capturing starts and a counter is shown to the player indicating how many poses have been captured.

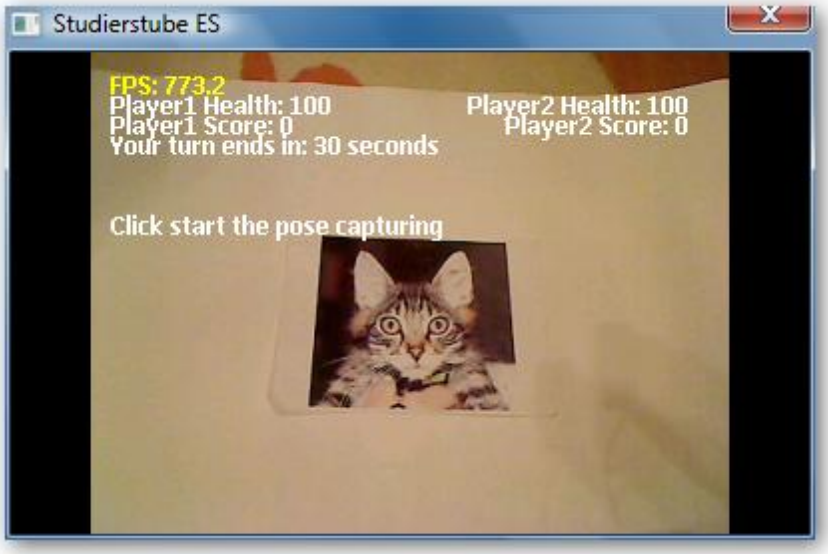

Figure 20: Game starting screen

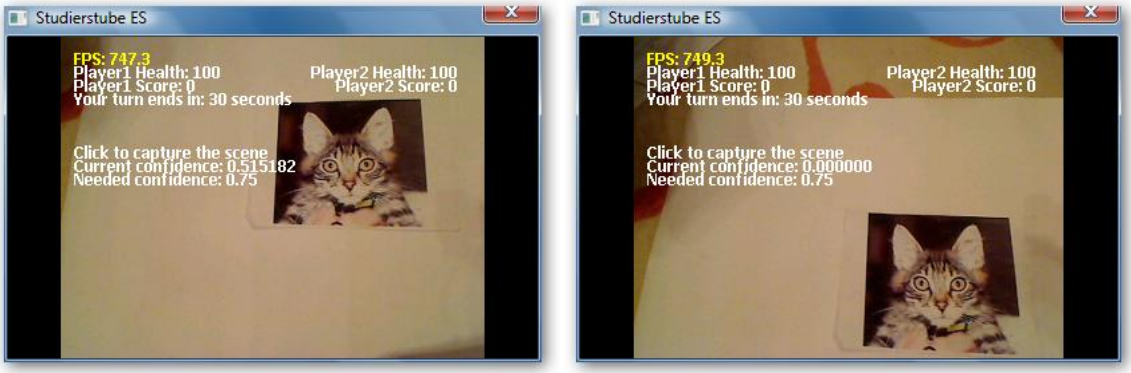

Figure 21: Game field capture. The confidence value indicates the coverage of the mosaic image

When this counter reaches 35, the user is asked to capture a few screenshots of the game field by clicking once on the screen on the phone. The user needs to reach the indicated confidence value (0.75) as can be seen in Figure 21. At this point, the game waits until the other player completes the same steps; the game starts as soon as both players have completed the screenshot capturing. Each player sees a scene similar to the one shown in Figure 22. The avatars are each located in random locations on opposite sides of the main tracking target.

Figure 22: Game starting screen showing the two players objects. The text below the player one score indicates if the player is active or not

The active player can select his or her avatar and move it anywhere within the game field, or draw a polygon. If a polygon is drawn, it is affected by gravity and falls to the bottom of the window; if it does not hit a player it can then be selected and thrown in any direction the player desires. If an avatar is hit by the thrown or falling polygon, it is pushed along the same movement direction and its health is reduced as can be seen in Figure 23. These steps repeat until one of the avatars loses all health points. If the health points of one of the players reach 0 the avatars are reset to random positions and a new round starts.

Figure 23: Player one has moved and thrown a stone at the other player. A new stone is being drawn. The stone has hit the opponent and it is being kicked out of the screen

6 Future Work

One area, in which further work could be done, would be the improvement of the sketch extraction. The current implementation is sensitive to different lighting conditions. The blob detection also produces false positives (sketches detected where there are none) so an increase of the accuracy in the sketch detection would benefit the game. Another area of improvement is the estimation of the marker poses to offer more accurate transformation matrices.

While designing the game, certain ideas were formulated but later not implemented due to time constraints. Allowing multiple polygons on the game field could be an interesting addition to the game, as well as extending it to more than two players. Random "unstable" sections of rock could be added around the playing field as virtual objects. When these are hit two things could happen:

- They could produce a few rocks which drop down. These rocks may be used by the avatars as ammunitions or they could hurt both avatars.
- They could reveal a hidden water well which starts raining down water. The water is hazardous to the avatars and if it submerges their figure they lose the game. There would be a limitation to two water wells in the game field to prevent the whole field getting flooded. If an avatar gets submerged by water it would lose the current round and the game would reset for the next round.

Another thing that could be added is the ability for players to construct ramps. These ramps could be constructed by drawing them on the phone. Each player would have a limited number of charges of ramps to draw. If a rock hits a ramp, it gets bounced back in the opposite direction or destroyed. The ramps could be used to deflect direct throws or to divert water away. Additional power-up items could be added to the game to modify either the stone projectiles or the avatars themselves. Power ups could be explosive stones to destroy the ramps, additional ramp charges, watertight ramps which would be impervious to water. Extending the gravitational influence to avatars could enable interesting new gameplay opportunities like jumping over obstacles, creating bottlenecks with ramps or diverting the water to the enemy avatar. Overall, we believe that the game presented in this thesis provides vast opportunities for future work, both on the technical challenges required to improve its robustness, and on the novel gameplay possibilities given by sketch-based interaction.

Acknowledgements

I would like firstly to thank my parents for all their support and help they have given me over the last few years. Secondly I would like to thank my supervisor Alessandro Mulloni for his aid and helpful suggestions given during the work on the thesis, and the very helpful comments during the reviewing of the thesis.

References

[1] Toon: Tom's object-oriented numerics library. http://www.edwardrosten.com/cvd/toon.html Retrieved: 23.10.2011

[2] Team 17 Worms. http://worms.team17.com/ Retrieved: 23.10.2011

[3] R. Azuma. A Survey of Augmented Reality. Presence: Teleoperators and Virtual Environments 6, 4 (August 1997), pages. 355 - 385.

[4] L. Barkhuus, M. Chalmers, P. Tennent, M. Hall, M. Bell, S. Sherwood, and B. Brown. Picking pockets on the lawn: The development of tactics and strategies in a mobile game. In *Proceedings of UbiComp 2005*, pages 358–374. Springer, 2005.

[5] S. Björk, J. Falk, R. Hansson, and P. Ljungstrand. Pirates! Using the physical world as a game board. In *IN PROCEEDINGS OF INTERACT 2001*, pages 9–13, 2001.

[6] S. Engelhardt, A. Langs, G. Lochmann, I. Schmidt, and S. Muller. Molearlert - an augmented reality game based on lemmings. In *Proc. 8th IEEE Int. Symp. Mixed and Augmented Reality ISMAR 2009*, pages 183–184, 2009.

[7] N. Hagbi, O. Bergig, J. El-Sana, and M. Billinghurst. Shape recognition and pose estimation for mobile augmented reality. 17(10):1369–1379, 2011.

[8] N. Hagbi, R. Grasset, O. Bergig, M. Billinghurst, and J. El-Sana. In-place sketching for content authoring in augmented reality games. In *Proc. IEEE Virtual Reality Conf. (VR)*, pages 91–94, 2010.

[9] A. Henrysson, M. Billinghurst, and M. Ollila. AR tennis. In *ACM SIGGRAPH 2006 Sketches*, Article No. 13 SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[10] I. Herbst, A. Braun, R. McCall, and W. Broll. Timewarp: Interactive time travel with a mobile mixed reality game. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, MobileHCI '08, pages 235–244, New York, NY, USA, 2008. ACM.

[11] D. T. Huynh, K. Raveendran, Y. Xu, K. Spreen, and B. MacIntyre. Art of defense: a collaborative handheld augmented reality board game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 135–142, New York, NY, USA, 2009. ACM.

[12] idSoftware. Quake. http://www.idsoftware.com/games/quake/quake# Retrieved 18.10.2011

[13] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. pages 282–292, 1994.

[14] A. Mulloni. A collaborative and location-aware application based on augmented reality for mobile devices. Master's thesis, Universit'a degli Studi di Udine, 2007.

[15] Revolutionary Concepts. UFO on Tape. http://www.revolutionaryconcepts.net/uot Retrieved: 23.10.2011

[16] A. Walker, R. Fisher, S. Perkins and E. Wolfart. Adaptive thresholding. http://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm Retrieved: 18.10.2011

[17] D. Schmalstieg and Daniel Wagner. Mobile phones as a platform for Augmented Reality. Proceedings of the IEEE VR 2008 Workshop on Software Engineering and Architectures for Realtime Interactive Systems (Reno, NV, USA), pp. 43-44, Shaker Verlag, March 2008.

[18] Sony Computer Entertainment. Eye of Judgement. http://www.eyeofjudgment.com/ Retrieved 23.10.2011

[19] Sony Computer Entertainment. EyePet. http://www.eyepet.com/ Retrieved 23.10.2011

[20] B. Thomas, B. Close, J. Donoghue, J. Squires, P. De Bondi, and W. Piekarski. First person indoor/outdoor augmented reality application: ARQuake. *Personal Ubiquitous Comput.*, 6:75–86, January 2002.

[21] THQ. Fantastic pets http://www.thq.com/us/fantastic-pets/360 Retrieved: 02.11. 2011.

[22] THQWireless. Star Wars Arcade: Falcon Gunner http://www.thq.com/uk/star-wars-flight-of-the-falcon/wireless Retrieved: 18.10.2011

[23] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Pose tracking from natural features on mobile phones. In *Proc. 7th IEEE/ACM Int. Symp. Mixed and Augmented Reality ISMAR 2008*, pages 125–134, 2008.

[24] D. Wagner and D. Schmalstieg. Muddleware for prototyping mixed reality multiuser games. In *Proc. IEEE Virtual Reality Conf. VR '07*, pages 235–238, 2007.

[25] D. Wagner, T. Pintaric, and D. Schmalstieg. The invisible train: a collaborative handheld augmented reality demonstrator. In *ACM SIGGRAPH 2004 Emerging technologies*, SIGGRAPH '04, page 12, New York, NY, USA, 2004. ACM.