Augmented Reality mirror game

Thijs Boumans Patrick Kramer Alexander Overvoorde
Tim van Rossum

June 26, 2015

Abstract

This report describes the development of an augmented reality game by students of the Delft University of Technology. This game, called the *Augmented Reality Mirror Game*, is a game that uses augmented reality technology to simulate lasers and objects with optical properties like mirrors and beam splitters. The goal of this game is to use these objects to route one or more laser beams from emitters to targets. Collaboration is an essential aspect of the game. Different players have distinct capabilities to manipulate the game world and solve the puzzles.

Contents

List of Figures

Chapter 1

Orientation

This chapter provides an overview of the orientation phase of the project, as well as giving an overview of the entire process of the project from start to finish. It shows an analysis of the project requirements, and the decisions that have been made during the project regarding choices of frameworks and libraries as well as game play elements. It also functions as a thorough introduction to this report.

For a more in-depth view on the research that has been done leading to these decisions, please refer to the Research Report in appendix A.

1.1 Project Description

While augmented reality research has grown into a mature field over the last years, the aspects of situational awareness and presence of augmented reality (AR) are still quite open research topics. This project is about designing and implementing a collaborative game to explore the different perception of situational awareness, presence and workload in a physical and an AR environment. The game is to be employed as an approximation of collaboratively solving complex problems, as they occur in crime scene investigation when using virtual co-location, i.e. remote crime scene experts to guide local investigators in AR to collaboratively analyze the crime scene.

1.2 Final Product

The goal of the game is to solve a puzzle by controlling laser beams using mirrors in such a way that a predefined target is hit. The game can be played by one or more local players and one or more remote players.

There are cards present for the local players that represent mirror bases. These must be placed on the table, which will be the locations for the mirrors. The local players will be able to see the mirrors they place through the use of AR technology. Each of the local players will only be given a few of the mirror bases needed to solve the puzzle, and as such solving it requires cooperation from all local players.

The remote players can also see the placed mirrors, and can rotate them to influence the path of the laser beam(s). Only by cooperation between local players (who can only move the mirror bases) and remote players (who can only rotate them) it becomes possible to hit the target and as such solve the puzzle.

The game provides various types of objects with different capabilities, allowing for more complex puzzles. Examples of such objects include, but are not limited to walls, laser beam splitters and checkpoints. For a complete overview of the game objects provided by the game, see section 2.3.

The game is designed to stimulate cooperation between physically co-located players and the remote player(s). It does so by dividing abilities required for solving the puzzles amongst all players as follows:

- Physically co-located players each get only a part of the mirror bases required to solve the puzzle, requiring input from all of these players.
- Physically remote players have the ability to rotate mirrors while the physically co-located players do not have this ability, requiring input from both physically co-located as well as remote players.

1.3 Software Design Methods

This section describes the design methods that were used during the project. It illustrates the methodology that was used to develop and coordinate the project during the development phase.

1.3.1 Design Process

In designing and implementing the product, it is important that requirements can be changed quickly and without much problems. This is not because the requirements are likely to change from the client side, but because the choice of AR technology may change over the course of the project because of technical issues. The available Virtual and Augmented Reality glasses are mostly still in development, and as such this may affect the technical viability of each device.

To deal with such changes, we use the Scrum methodology [Sutherland and Schwaber, 2013]. This methodology describes a set of rules that, amongst others, makes it easier to deal with various changes during the development process.

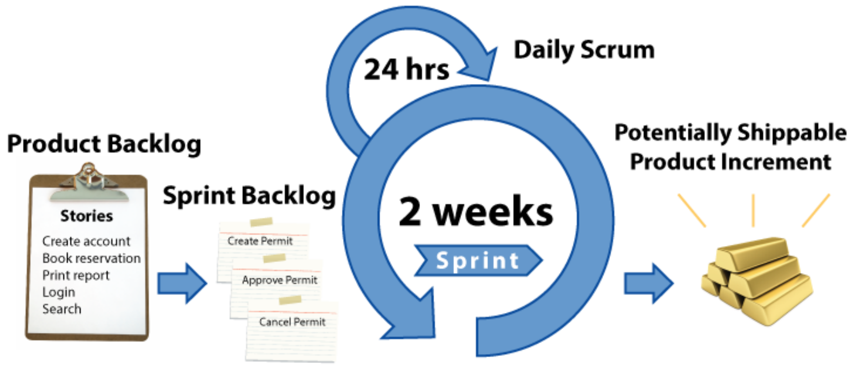


Figure 1.1: Scrum methodology overview

The methodology in graph form can be seen in 1.1. The methodology shown is nearly identical to what we use, the only exception being that our sprints only last one week instead of two.

1.3.2 Organization

To be able to simultaneously work on the project without conflicts, we use Git as a version control system. The project is stored remotely on GitHub, ensuring the work is efficiently shared between all team members. Also, Git stores all commits that have been done. These can be reviewed on GitHub, and it is possible to go back to a commit stage if we absolutely need to, in case of something horribly wrong happening to the project.

To coordinate and divide the tasks, as well as to maintain the items in the Scrum backlog, we use Trello. Trello is an on-line service that provides a dynamic way to organize items in various lists. It does so by using cards as bullet points in a list. It is also possible to assign certain people to those cards, which can be used to visibly divide tasks among the group members. Using labels, each card can also be categorized as a task relating to a certain part or parts of the project, such as software engineering, graphics design, networking, etc. We created lists to keep track of which items were in the backlog, which items were being worked on and which items were already done. This allowed us to easily see what was being done and what was done.

The project is licensed under the terms of the MIT license. We chose this license because it allows other developers to learn from this project. Additionally, since this project is done as a part of a research project, we believe making it open source may help future researchers in the same field. The full terms of the MIT license can be found at `http://opensource.org/licenses/mit-license.html`

To ensure the C# source code in the project meets common coding standards (as set by Microsoft), we use the code analysis tools FxCop and StyleCop in combination with SonarQube. FxCop performs static code analysis, like code complexity and some naming conventions. StyleCop, on the other hand, focuses more on code style which includes use of spacing and documentation as well as other factors. SonarQube is a platform that unifies the reports from these tools and provides a clean overview of the combined issues found by FxCop and StyleCop, as well as some simple metrics SonarQube has built-in. Further detail about maintaining code quality is described in the QA section of the report, chapter 4.

1.3.3 Design Architecture

Because the product is a game and the goal of the project is more focused on the game mechanics rather than the underlying engine, we chose to use Unity as a starting point. Unity provides a platform independent development environment for creating games, and offers many features commonly used in games.

Using Unity means that the project architecture is bound to the loosely coupled component-based architecture that Unity provides, although it is possible to include principles from object-oriented programming to some extent.

1.4 Process

This section describes the process of the project. The different phases during the project are highlighted in this section.

1.4.1 Early Preparations

Before the project started, we had a meeting halfway through March with our coach about what the project entails and what is currently possible, given the hardware that we have today. After a brainstorm session and a pitch session with our coach and client, we were shown what kind of hardware the TU Delft has available, and also what the limitations of this hardware are. In this phase, a product plan was also created. This document describes the planning for the duration of the project, and can be found in appendix B.

1.4.2 Research

The first two weeks were the main research phase. This entailed that the research report had to be written. The second week was partly devoted to writing the report, and also to testing out more functionality of the software and hardware. The first game object models were also developed during this time, and there were plans for a first demo at the end of week 3 or at the beginning of week 4. The research report, which was the result of these two weeks, can be found in appendix A

1.4.3 Programming the basic game

The next two weeks revolved around creating game objects and game play. After some testing with markers and AR glasses, as well game objects, a first demo was also developed during this time. This phase also saw unit testing of game elements, as well as heavy usage of StyleCop and SonarQube to clean up respectively reorganize code in order to deliver clean code to SIG, for our first submission.

1.4.4 The OpenCV server

Week 5 finally saw the first demo being demonstrated to the coach. The coach was satisfied, but there was a lot to be done before the project could be considered finished. During this week, networking was revamped and development of a server that makes use of the computer vision library OpenCV started. This decision was based on various technical challenges we came across during development. The reason for this decision can be found in the Implementation chapter (chapter 3).

1.4.5 Restructuring the entire project

Week 6 began with a massive restructuring of the project. Right before the code was to be handed in to SIG, Unity failed to build the project completely. There were no errors in the scripts, but the Unity compiler kept throwing unexplainable error messages. This caused us to move everything that we wanted to move from the original project to a new Unity project.

Even though this caused a setback in the project planning, this issue gave us an opportunity to take a critical look at our code base. After completing this task and handing in the package for SIG, development could continue on the OpenCV server. The results for the SIG evaluation from this week can be found in appendix C.

1.4.6 The new projection code

Week 7 began with further development on the OpenCV server and the code needed for correct projection of the META One. The projection code base was overhauled on Monday, mainly to allow for better unit testing. During this week, the midterm meeting also took place, and we could show off what we had up until that point. The coach and the customer(s) (Stephan could not be there, so he sent some of his colleagues to check in on how the project went) were impressed, but they also said that development had to continue for the time being. At the end of week 7, the server was completed, but now the true challenge began: integrating all the software into one single product.

Week 8 started with the integration of all the parts of the software. From the start on, this proved quite challenging. Players could rotate mirrors by rotating the markers (which shouldn't happen, as this would kill off the co-operative element of the game), as well as other things. As such, development on projection code was once again necessary. Throughout this week, massive progress was made towards integration of the various parts of the software, especially from early Wednesday on. Because of the progress considering integration of software, level design could finally start.

1.4.7 Further integration

Week 9 was even further devoted to integration of all the various bits and pieces of the software. Once again, massive progress was made in integrating all software, and at the end of the week, the first fully working version was finally released.

1.4.8 Wrapping up

At the end of week 10, the final report had to be handed in. This meant that the project had to be wrapped up. The main activity in this week was testing with other users (the client and a few volunteers), as well as checking the final report before it had to be handed in. Also, the info sheet was written in this week, as it is required to have that checked before the final report can be handed in.

1.4.9 The process table

The following table gives a nice and short overview regarding what was done every week. An X in a particular subproject and week means that, during that week, that subproject was developed further.

Weeks	1	2	3	4	5	6	7	8	9	10
Research	X	X								
Final report				X	X	X	X	X	X	X
Networking	X	X			X	X	X		X	X
Augmented reality				X	X	X	X	X		
Game elements	X	X	X	X						
Testing			X	X	X	X	X	X		X
Projection				X	X	X	X	X	X	
Level design								X	X	X

Chapter 2

Design

This chapter explains the design of the system. This includes: back-end design, model/graphical design, and main activities.

2.1 Main activities

There are several main activities in the system, corresponding to the two main user types of the system. These user types are both the physically co- located players as well as the physically remote players. The activities are as follows:

2.1.1 A local user wants to start a game

The local user ensures the main camera is set up according to the provided guidelines. The local user then starts the server on the machine the camera is connected to. The local user needs to take note of the IP address of the server machine and fill this into his or her local machine (to which the Meta One glasses are attached). The other players also fill in the same IP address. The server can also run on one of the players' machines. The local user needs to wait for others to join the game. When at least two local players and at least one remote player has joined the game, the game will start.

2.1.2 A local user wants to join a game

A local player wants to join an active game. Before this can happen, a game has to be started first. The local player needs to acquire the IP address of the server machine and fill this in. Once the game starts, the local player will see virtual objects projected through the Meta One glasses.

2.1.3 A remote user wants to join a game

A physically remote player wants to join an active game. Before this can happen, a game has to be started first. The remote user needs to acquire the IP address of the local server machine (see 2.1.1) and fill this in. The remote user will see a virtual version of the same game world as the local players.

2.1.4 A local player wants to move a mirror

A local player is partaking in an active game. To hit the target, they need to move a mirror to a certain point to allow the laser beam coming from the emitter to be deflected and therefore to hit the target. The local player uses movable markers to place a mirror in the game world. Using the Meta One glasses, local players can see the game objects when at least one marker is in their view. For this example, it is assumed that the mirror is rotated such that no more rotation is required. The local player can then take the mirror marker, move it to the point it should be moved to, and complete the game that way.

2.1.5 A remote player wants to rotate a mirror

Remote players, due to them not being physically co-located with the local players, cannot move mirror markers, as that would require them to meet up with the local players and to play with them. However, remote players have access to an ability that the local players do not have access to, that being the ability to rotate selected mirrors. The remote player uses a computer for this. In order to rotate a mirror, the player can click on a mirror to select and rotate it. When a mirror is selected, the outer frame of the mirror (shown in 2.12) will change color from gold to bright yellow, indicating that this mirror is selected (the color change will only be visible for the remote player responsible for clicking on that mirror, even other remote players will not be able to see this color change). The selected mirror can be rotated clockwise using the right mouse button, and counter-clockwise using the left mouse button.

2.1.6 Use case diagrams

To illustrate the given main activities, two use case diagrams are displayed here. The use cases for the remote player are depicted in 2.1, while the use cases for the local player are depicted in 2.2.

2.2 Back-end design

The system is composed of several main parts. For this purpose, we divided the C# code over various components, which will be highlighted below.

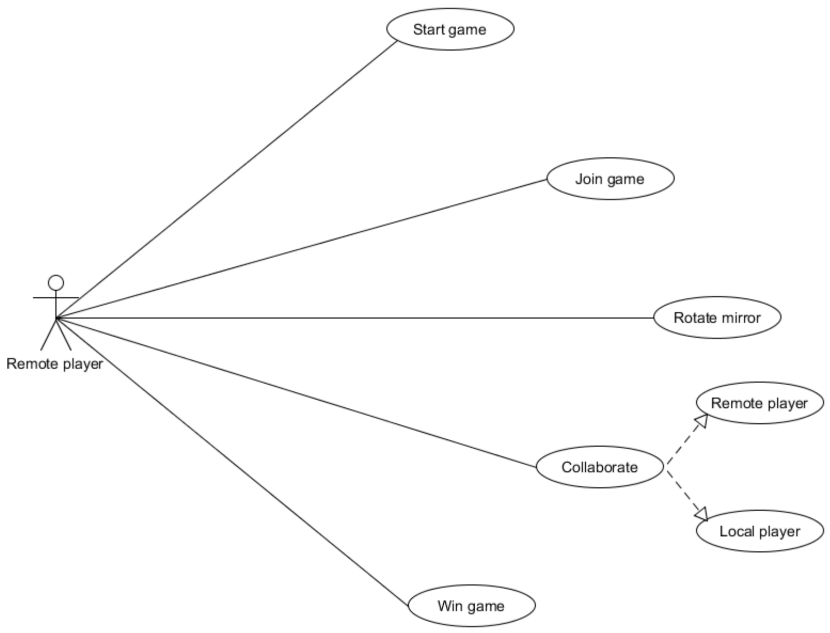


Figure 2.1: The use cases for the remote player.

2.2.1 The Core component

The `Core` component contains all the code for all the core game play elements. The term ”core game play elements” refers to all objects necessary to play the basic game with, basically the code for the game objects. Using only the code from the `Core` component, it would be possible to create the same game without using AR technology or using markers to play the game with, for that matter.

The `Core` component has two subcomponents: The `Core.Emitter` component, which contains code for objects that emit laser beams, and the `Core.Receiver` component, which contains code for objects that receive laser beams and have to do something with these beams.

There are several game objects that emit laser beams as well as receive them (as seen in 2.3). This causes the `Core.Emitter` and `Core.Receiver` components to have some coupling. However, the coupling is minor, and it only exists between sub-components in the same main component. As such, this should not cause that much of a problem, when considered in a software engineering context.

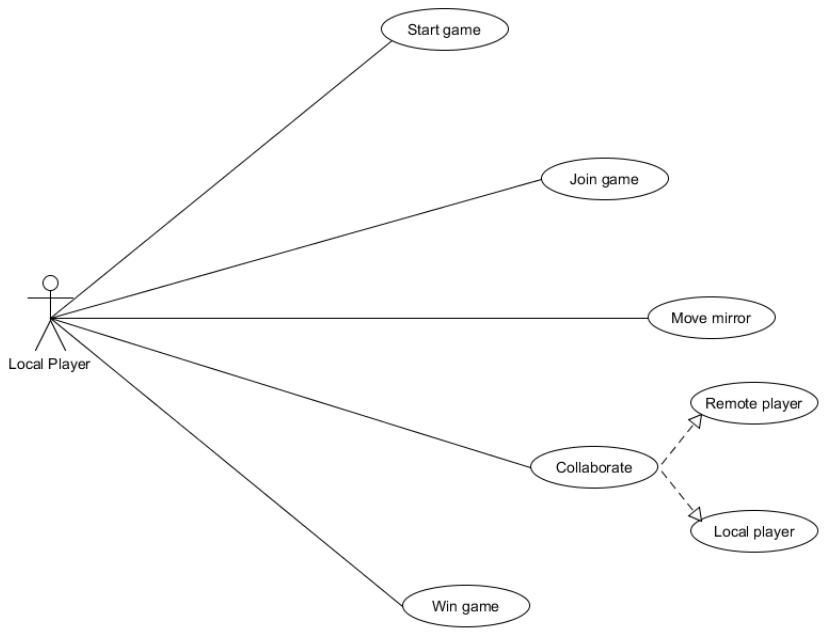


Figure 2.2: The use cases for the local player.

2.2.2 The Network component

The `Network` component has undergone several changes since the start of the project. These changes are described in this part.

The first take on creating a network functionality in the game was to use the standard way of networking within Unity itself. This involved creating a rather simple network script that allowed the game to connect to a Unity master server, which then handled all changes that were recorded by network view components of objects, and sent them to all the players, synchronizing the game world. There are several tutorials on how to write such a networking script and how to use it in a multi-player game, the tutorial that was used for the script in the component can be found here: `http://www.paladinstudios.com/2013/07/10/how-to-create-an-online-multiplayer-game-with-unity/`.

Unfortunately, due to the major differences between local and remote players, as well as the way the Meta One manipulated object positions and rotations to fit the real world, we needed a custom solution to be able to synchronize the world state in a more controlled way. Because of this, a different solution was required to create synchronization across the game world. The next option was to use the Photon Unity Networking package, which not only supported multi-player better (it allowed for more people to play the same game), but it also allowed VR and AR games to have multi-player functionality (there is a free

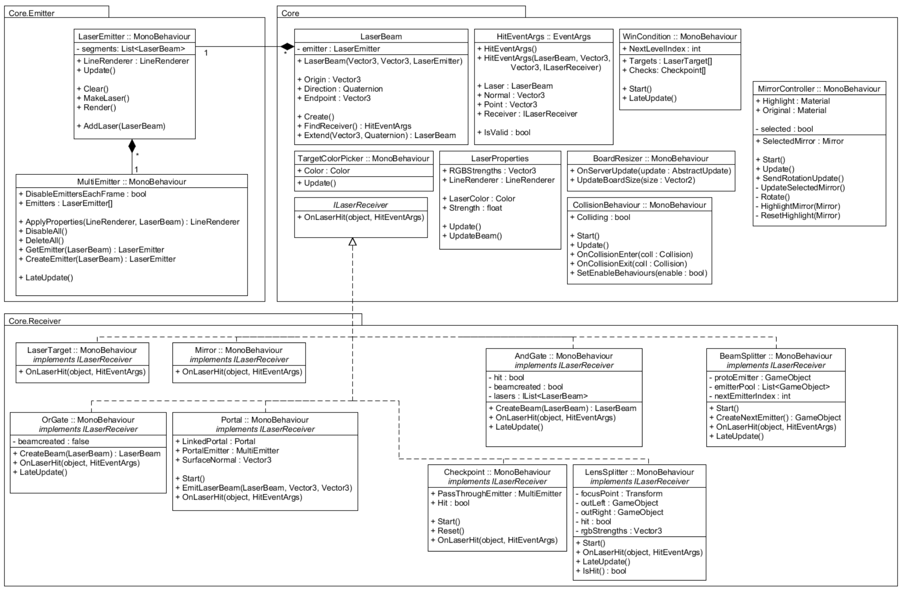

Core.Emitter
LaserEmitter :: MonoBehaviour
- segments: List<LaserBeam>
+ LineRenderer : LineRenderer
+ Update()
+ Clear()
+ MakeLaser()
+ Render()
+ AddLaser(LaserBeam)
1
*
*
1
MultiEmitter :: MonoBehaviour
+ DisableEmittersEachFrame : bool
+ Emitters : LaserEmitter[]
+ ApplyProperties(LineRenderer, LaserBeam) : LineRenderer
+ DisableAll()
+ DeleteAll()
+ GetEmitter(LaserBeam) : LaserEmitter
+ CreateEmitter(LaserBeam) : LaserEmitter
+ LateUpdate()
Core
LaserBeam
- emitter : LaserEmitter
+ LaserBeam(Vector3, Vector3, LaserEmitter)
+ Origin : Vector3
+ Direction : Quaternion
+ Endpoint : Vector3
+ Create()
+ FindReceiver() : HitEventArgs
+ Extend(Vector3, Quaternion) : LaserBeam
HitEventArgs :: EventArgs
+ HitEventArgs()
+ HitEventArgs(LaserBeam, Vector3, Vector3, ILaserReceiver)
+ Laser : LaserBeam
+ Normal : Vector3
+ Point : Vector3
+ Receiver : ILaserReceiver
+ IsValid : bool
WinCondition :: MonoBehaviour
+ NextLevelIndex : int
+ Targets : LaserTarget[]
+ Checks: Checkpoint[]
+ Start()
+ LateUpdate()
MirrorController :: MonoBehaviour
+ Highlight : Material
+ Original : Material
- selected : bool
+ SelectedMirror : Mirror
+ Start()
+ Update()
+ SendRotationUpdate()
- UpdateSelectedMirror()
- Rotate()
- HighlightMirror(Mirror)
- ResetHighlight(Mirror)
TargetColorPicker :: MonoBehaviour
+ Color : Color
+ Update()
ILaserReceiver
+ OnLaserHit(object, HitEventArgs)
LaserProperties
+ RGBStrengths : Vector3
+ LineRenderer : LineRenderer
+ LaserColor : Color
+ Strength : float
+ Update()
+ UpdateBeam()
BoardResizer :: MonoBehaviour
+ OnServerUpdate(update : AbstractUpdate)
+ UpdateBoardSize(size : Vector2)
CollisionBehaviour :: MonoBehaviour
+ Colliding : bool
+ Start()
+ Update()
+ OnCollisionEnter(coll : Collision)
+ OnCollisionExit(coll : Collision)
+ SetEnableBehaviours(enable : bool)
Core.Receiver
LaserTarget :: MonoBehaviour
implements ILaserReceiver
+ OnLaserHit(object, HitEventArgs)
Mirror :: MonoBehaviour
implements ILaserReceiver
+ OnLaserHit(object, HitEventArgs)
AndGate :: MonoBehaviour
implements ILaserReceiver
- hit : bool
- beamcreated : bool
- lasers : IList<LaserBeam>
+ CreateBeam(LaserBeam) : LaserBeam
+ OnLaserHit(object, HitEventArgs)
+ LateUpdate()
BeamSplitter :: MonoBehaviour
implements ILaserReceiver
- protoEmitter : GameObject
- emitterPool : List<GameObject>
- nextEmitterIndex : int
+ Start()
+ CreateNextEmitter() : GameObject
+ OnLaserHit(object, HitEventArgs)
+ LateUpdate()
OrGate :: MonoBehaviour
implements ILaserReceiver
- beamcreated : false
+ CreateBeam(LaserBeam) : LaserBeam
+ OnLaserHit(object, HitEventArgs)
+ LateUpdate()
Portal :: MonoBehaviour
implements ILaserReceiver
+ LinkedPortal : Portal
+ PortalEmitter : MultiEmitter
+ SurfaceNormal : Vector3
+ Start()
+ EmitLaserBeam(LaserBeam, Vector3, Vector3)
+ OnLaserHit(object, HitEventArgs)
Checkpoint :: MonoBehaviour
implements ILaserReceiver
+ PassThroughEmitter : MultiEmitter
+ Hit : bool
+ Start()
+ Reset()
+ OnLaserHit(object, HitEventArgs)
LensSplitter :: MonoBehaviour
implements ILaserReceiver
- focusPoint : Transform
- outLeft : GameObject
- outRight : GameObject
- hit : bool
- rgbStrengths : Vector3
+ Start()
+ OnLaserHit(object, HitEventArgs)
+ LateUpdate()
+ IsHit() : bool

Figure 2.3: Class diagram of the `Core` component

package available on the Unity Asset Store, which can be found here: `https://www.assetstore.unity3d.com/en/#!/content/1786`). However, this was not used, as the problem was not that we needed to connect several AR players in the game world (as these would be the local players, and therefore would be able to see what was going on in the game world by just looking at the marker that was placed near them). As such, a different solution had to be found.

The final solution we used is as follows: Instead of relying on Unity for the networking functionality, we decided to combine this problem with another issue we came across regarding the detection and synchronization of marker locations. This solution was to deploy a server, written in C++, that uses OpenCV to detect the markers. This also allows us full control over the detection and manipulation of object transformations in Unity, as this was causing issues with both the Meta One SDK as well as Vuforia.

The OpenCV server makes use of a single camera to detect marker locations. These locations are then sent to all clients over a socket connection. This solution makes it easier to synchronize object positions between users because all game activities take place in a predefined area on a table, and the users need only look at a single marker to be able to see the entire scene.

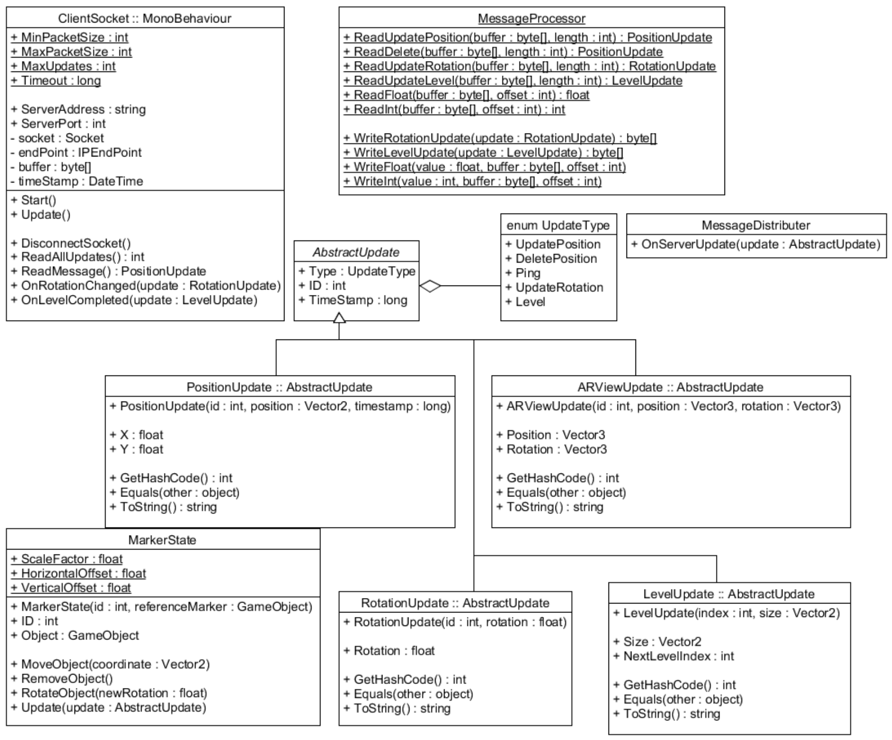

ClientSocket :: MonoBehaviour
+ MinPacketSize : int
+ MaxPacketSize : int
+ MaxUpdates : int
+ Timeout : long
+ ServerAddress : string
+ ServerPort : int
- socket : Socket
- endPoint : IPEndPoint
- buffer : byte[]
- timeStamp : DateTime
+ Start()
+ Update()
+ DisconnectSocket()
+ ReadAllUpdates() : int
+ ReadMessage() : PositionUpdate
+ OnRotationChanged(update : RotationUpdate)
+ OnLevelCompleted(update : LevelUpdate)
MessageProcessor
+ ReadUpdatePosition(buffer : byte[], length : int) : PositionUpdate
+ ReadDelete(buffer : byte[], length : int) : PositionUpdate
+ ReadUpdateRotation(buffer : byte[], length : int) : RotationUpdate
+ ReadUpdateLevel(buffer : byte[], length : int) : LevelUpdate
+ ReadFloat(buffer : byte[], offset : int) : float
+ ReadInt(buffer : byte[], offset : int) : int
+ WriteRotationUpdate(update : RotationUpdate) : byte[]
+ WriteLevelUpdate(update : LevelUpdate) : byte[]
+ WriteFloat(value : float, buffer : byte[], offset : int)
+ WriteInt(value : int, buffer : byte[], offset : int)
AbstractUpdate
+ Type : UpdateType
+ ID : int
+ TimeStamp : long
enum UpdateType
+ UpdatePosition
+ DeletePosition
+ Ping
+ UpdateRotation
+ Level
MessageDistributer
+ OnServerUpdate(update : AbstractUpdate)
PositionUpdate :: AbstractUpdate
+ PositionUpdate(id : int, position : Vector2, timestamp : long)
+ X : float
+ Y : float
+ GetHashCode() : int
+ Equals(other : object)
+ ToString() : string
ARViewUpdate :: AbstractUpdate
+ ARViewUpdate(id : int, position : Vector3, rotation : Vector3)
+ Position : Vector3
+ Rotation : Vector3
+ GetHashCode() : int
+ Equals(other : object)
+ ToString() : string
MarkerState
+ ScaleFactor : float
+ HorizontalOffset : float
+ VerticalOffset : float
+ MarkerState(id : int, referenceMarker : GameObject)
+ ID : int
+ Object : GameObject
+ MoveObject(coordinate : Vector2)
+ RemoveObject()
+ RotateObject(newRotation : float)
+ Update(update : AbstractUpdate)
RotationUpdate :: AbstractUpdate
+ RotationUpdate(id : int, rotation : float)
+ Rotation : float
+ GetHashCode() : int
+ Equals(other : object)
+ ToString() : string
LevelUpdate :: AbstractUpdate
+ LevelUpdate(index : int, size : Vector2)
+ Size : Vector2
+ NextLevelIndex : int
+ GetHashCode() : int
+ Equals(other : object)
+ ToString() : string

Figure 2.4: Class diagram of the `Network` component

2.2.3 The Projection component

The `Projection` component is responsible for projecting the world to the Meta One glasses. It takes care of detecting markers in the playing area and projects all objects relative to those markers.

The main responsibility, fixing the projection to the Meta One glasses, is partly taken care of by the Meta SDK. However, the Meta SDK moves and rotates all game objects to fit the actual position and rotation of the Meta One. Due to the limited field of vision of the Meta One glasses, not all markers can be seen at the same time. In order to ensure the game objects and corresponding laser beams still appear at the correct positions, the Network component provides relative positions and rotations for all game objects in the playing area (also see paragraph 2.2.2). The `Projection` component then uses this information to place the objects in correct positions relative to any marker detected by the Meta SDK. This ensures that all objects remain visible and in the correct locations, as long as the Meta SDK can see at least a single marker.

Note that even if the marker cannot be detected temporarily, there is only a slight error in the locations of objects. This is due to the use of SLAM localization for tracking markers. See the paragraph about SLAM localization (3.1.1) for details on how SLAM localization works.

See figure 2.5 for the class diagram for the Projection component.

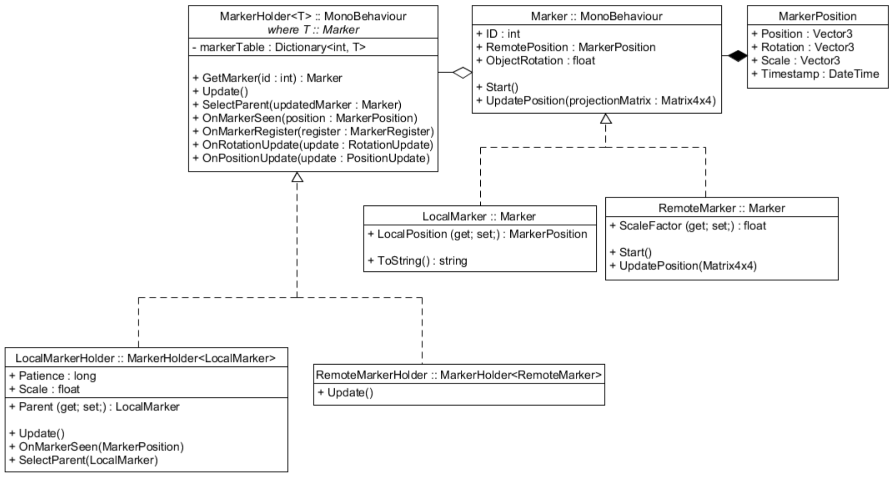

MarkerHolder<T> :: MonoBehaviour
where T :: Marker
- markerTable : Dictionary<int, T>
+ GetMarker(id : int) : Marker
+ Update()
+ SelectParent(updatedMarker : Marker)
+ OnMarkerSeen(position : MarkerPosition)
+ OnMarkerRegister(register : MarkerRegister)
+ OnRotationUpdate(update : RotationUpdate)
+ OnPositionUpdate(update : PositionUpdate)
Marker :: MonoBehaviour
+ ID : int
+ RemotePosition : MarkerPosition
+ ObjectRotation : float
+ Start()
+ UpdatePosition(projectionMatrix : Matrix4x4)
MarkerPosition
+ Position : Vector3
+ Rotation : Vector3
+ Scale : Vector3
+ Timestamp : DateTime
LocalMarker :: Marker
+ LocalPosition (get; set;) : MarkerPosition
+ ToString() : string
RemoteMarker :: Marker
+ ScaleFactor (get; set;) : float
+ Start()
+ UpdatePosition(Matrix4x4)
LocalMarkerHolder :: MarkerHolder<LocalMarker>
+ Patience : long
+ Scale : float
+ Parent (get; set;) : LocalMarker
+ Update()
+ OnMarkerSeen(MarkerPosition)
+ SelectParent(LocalMarker)
RemoteMarkerHolder :: MarkerHolder<RemoteMarker>
+ Update()

Figure 2.5: Class diagram of the `Projection` component

2.2.4 The Vision component

The `Vision` component is a small component responsible for providing hardware-specific details regarding marker tracking. It used to be a part of the `Projection` component, but it got split off, because it had little to do with the projection of the game objects into the world, and more with tracking the actual markers. Also, the `Projection` component became way too big during development: it started with about four to five classes, and those got split up into about fifteen classes, mainly because they violated software engineering principles such as the single responsibility principle. Because it got so big, some of the classes needed to be split off from the component and put into another component.

By design, and because of how it was conceived, the `Vision` component depends lightly on the `Projection` component. The only communication from the `Vision` component to the `Projection` component is by sending out messages into the object hierarchy, which are then received by components that can actually receive said messages.

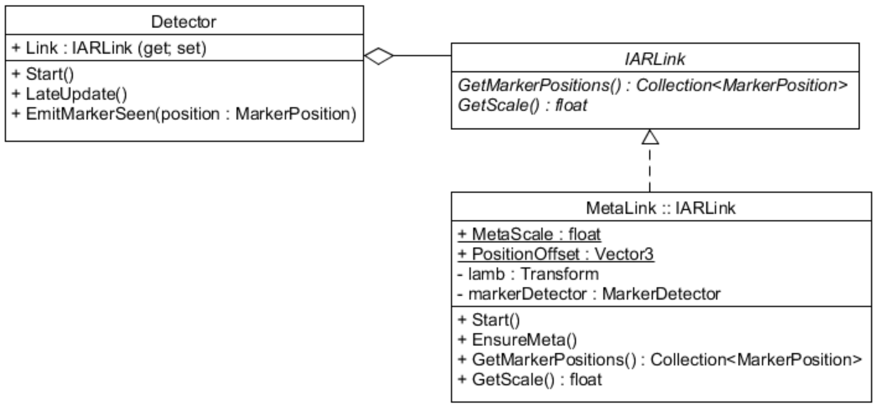

Detector
+ Link : IARLink (get; set)
+ Start()
+ LateUpdate()
+ EmitMarkerSeen(position : MarkerPosition)
IARLink
GetMarkerPositions() : Collection<MarkerPosition>
GetScale() : float
MetaLink :: IARLink
+ MetaScale : float
+ PositionOffset : Vector3
- lamb : Transform
- markerDetector : MarkerDetector
+ Start()
+ EnsureMeta()
+ GetMarkerPositions() : Collection<MarkerPosition>
+ GetScale() : float

Figure 2.6: Class diagram of the `Vision` component

2.2.5 The Level component

The `Level` component contains the functionality to load and build levels for the players to solve. It contains a way to load levels created using a map editor application called Tiled [Lindeijer, 2014]. Using this map editor, it becomes possible to create levels fairly easily using a visual editor. See figure 2.7 for an example level opened in the Tiled editor. The tiles representing various game objects can be seen in the lower right of the window, and the level itself is displayed in the center.

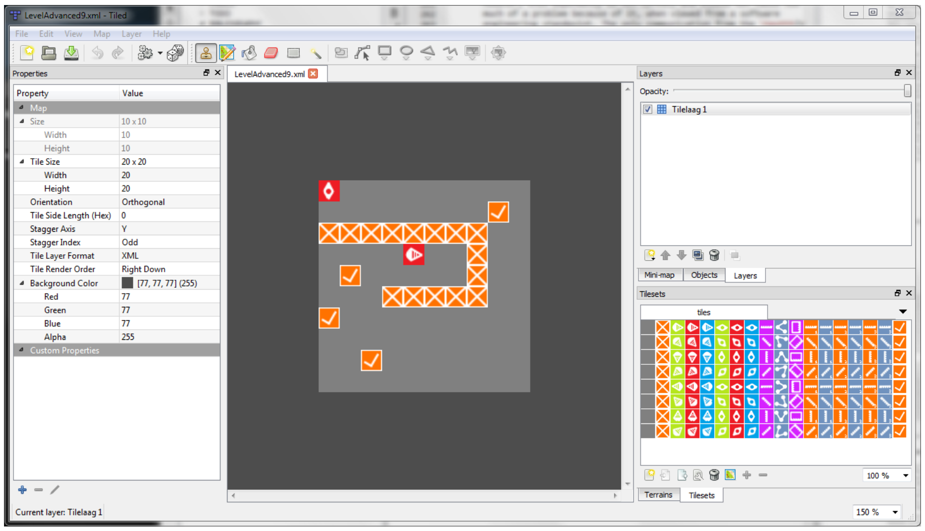
2.2.6 The Graphics component

The `Graphics` component provides graphical functionality to the game. Since most graphics-related functions are provided by Unity, this component is very small, containing only very limited functionality that Unity did not already provide. The main function included in this component is the code that draws the laser beam.

2.2.7 Experimental code: the RandomLevel component

The `RandomLevel` component is an abandoned and deleted component, which contained code that, when deployed, would create rather simple but random levels with one target and one laser. The levels are always solvable, and would be created according to an algorithm that is explained in this section.

The algorithm that was used to create random levels is as follows: First, a square grid of points is made. The grid was represented by 2D array (a matrix)



Property	Value
Map	
Size	10 x 10
Width	10
Height	10
Tile Size	20 x 20
Width	20
Height	20
Orientation	Orthogonal
Tile Side Length (Hex)	0
Stagger Axis	Y
Stagger Index	Odd
Tile Layer Format	XML
Tile Render Order	Right Down
Background Color	[77, 77, 77] (255)
Red	77
Green	77
Blue	77
Alpha	255
Custom Properties	

Current layer: Tilelaag 1

Figure 2.7: Tiled Map Editor Window

of grid points. Second, the laser target is placed in the middle of the matrix. Third, a spiral path is created that goes from the laser to an outer edge of the matrix. This path is then known as the critical path. Finally, walls are added to the level randomly, however they are never placed on either the target, a part of the critical path, or the laser. The walls are also rotated either horizontally or vertically. As can be seen, the levels created were rather simple (as they did not contain our more advanced game objects, only targets and walls), and solving them could always be done with several mirrors. The four phases of the randomization algorithm are displayed graphically in 2.10. The first image depicts the grid, the second depicts the laser target as a red dot in the middle of the map, the third image depicts the spiral path that is created from the target (the orange dot) to the edge of the map (this spiral path always bends four times, and can start in any random direction), the final image depicts the random placement of walls as purple dots.

The random level generation was added very early on in the project, and there was some value in developing it further (think of replay value for the game, for example). It was also changed later to allow for more possible paths (not only spirals, but also paths with less or more than four bends). However, it was scrapped after discussing it with our coach, after which we agreed that the increase of the replay value of the game was not worth it. Another reason to abandon it was because the classes belonging to it were some of the most complex in the project. Even after several attempts to rewrite these classes, they were still just barely under the maximum complexity that we allowed for our classes. This was measured with SonarQube, explained in section 4.

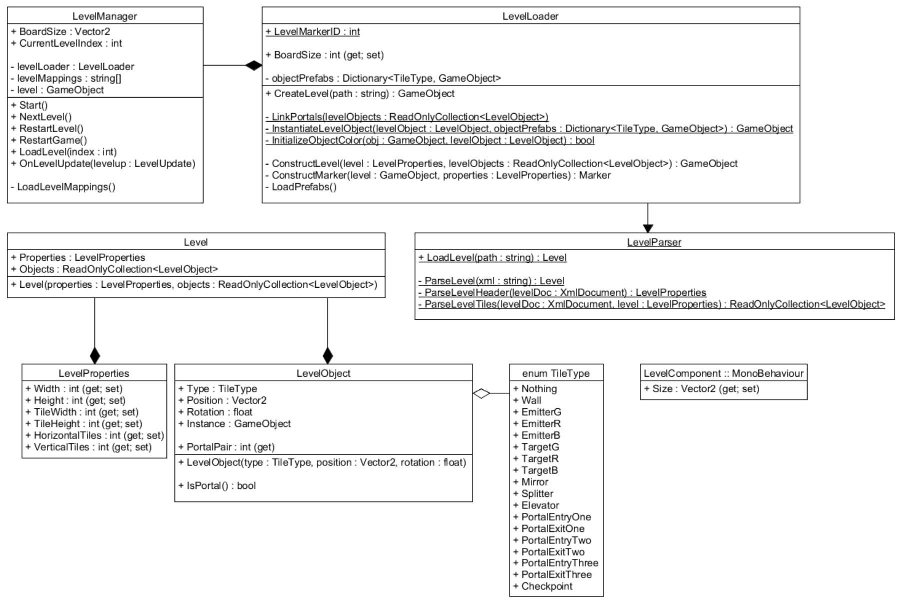

LevelManager
+ BoardSize : Vector2
+ CurrentLevelIndex : int
- levelLoader : LevelLoader
- levelMappings : string[]
- level : GameObject
+ Start()
+ NextLevel()
+ RestartLevel()
+ RestartGame()
+ LoadLevel(index : int)
+ OnLevelUpdate(levelup : LevelUpdate)
- LoadLevelMappings()
LevelLoader
+ LevelMarkerID : int
+ BoardSize : int (get; set)
- objectPrefabs : Dictionary<TileType, GameObject>
+ CreateLevel(path : string) : GameObject
- LinkPortals(levelObjects : ReadOnlyCollection<LevelObject>)
- InstantiateLevelObject(levelObject : LevelObject, objectPrefabs : Dictionary<TileType, GameObject>) : GameObject
- InitializeObjectColor(obj : GameObject, levelObject : LevelObject) : bool
- ConstructLevel(level : LevelProperties, levelObjects : ReadOnlyCollection<LevelObject>) : GameObject
- ConstructMarker(level : GameObject, properties : LevelProperties) : Marker
- LoadPrefabs()
Level
+ Properties : LevelProperties
+ Objects : ReadOnlyCollection<LevelObject>
+ Level(properties : LevelProperties, objects : ReadOnlyCollection<LevelObject>)
LevelParser
+ LoadLevel(path : string) : Level
- ParseLevel(xml : string) : Level
- ParseLevelHeader(levelDoc : XmlDocument) : LevelProperties
- ParseLevelTiles(levelDoc : XmlDocument, level : LevelProperties) : ReadOnlyCollection<LevelObject>
LevelProperties
+ Width : int (get; set)
+ Height : int (get; set)
+ TileWidth : int (get; set)
+ TileHeight : int (get; set)
+ HorizontalTiles : int (get; set)
+ VerticalTiles : int (get; set)
LevelObject
+ Type : TileType
+ Position : Vector2
+ Rotation : float
+ Instance : GameObject
+ PortalPair : int (get)
+ LevelObject(type : TileType, position : Vector2, rotation : float)
+ IsPortal() : bool
enum TileType
+ Nothing
+ Wall
+ EmitterG
+ EmitterR
+ EmitterB
+ TargetG
+ TargetR
+ TargetB
+ Mirror
+ Splitter
+ Elevator
+ PortalEntryOne
+ PortalExitOne
+ PortalEntryTwo
+ PortalExitTwo
+ PortalEntryThree
+ PortalExitThree
+ Checkpoint
LevelComponent :: MonoBehaviour
+ Size : Vector2 (get; set)

Figure 2.8: Class diagram of the `Level` component

2.3 Game elements

For designing the 3D models, we used Blender. Blender is a free application for 3D modeling, under the GNU General Public License, and Unity natively supports Blender models (provided it is installed on the system). We have chosen for a light looking style featuring nature inspired models and gold and crystal based materials. The light modeling style causes slight misalignments with the ground to be less noticeable and makes the lack of feedback from moving a card feel less odd. The crystals and gold just feels good in combination with the beams of light.

The following sections display and describe the graphics used in the game play elements, as well as the function of these elements.

2.3.1 Laser target

The laser target is the main target of the game. It consists of a small container, which contains a crystal. The point of the game is to direct a laser beam from an emitter to this target. When the target is hit by a laser beam, the outer columns around the crystal inside will rotate and spread out, indicating that the target has been hit. The game will proceed to the next level once all targets are hit, provided that the level contains no checkpoints. An image of the target is

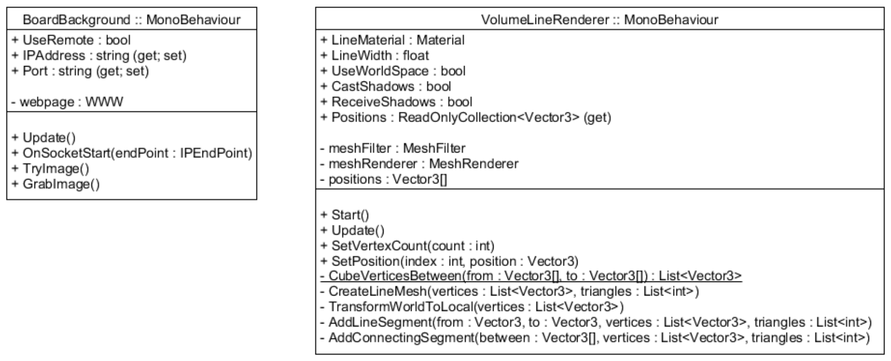

BoardBackground :: MonoBehaviour
+ UseRemote : bool
+ IPAddress : string (get; set)
+ Port : string (get; set)
- webpage : WWW
+ Update()
+ OnSocketStart(endPoint : IPEndPoint)
+ TryImage()
+ GrabImage()
VolumeLineRenderer :: MonoBehaviour
+ LineMaterial : Material
+ LineWidth : float
+ UseWorldSpace : bool
+ CastShadows : bool
+ ReceiveShadows : bool
+ Positions : ReadOnlyCollection<Vector3> (get)
- meshFilter : MeshFilter
- meshRenderer : MeshRenderer
- positions : Vector3[]
+ Start()
+ Update()
+ SetVertexCount(count : int)
+ SetPosition(index : int, position : Vector3)
- CubeVerticesBetween(from : Vector3[], to : Vector3[]) : List<Vector3>
- CreateLineMesh(vertices : List<Vector3>, triangles : List<int>)
- TransformWorldToLocal(vertices : List<Vector3>)
- AddLineSegment(from : Vector3, to : Vector3, vertices : List<Vector3>, triangles : List<int>)
- AddConnectingSegment(between : Vector3[], vertices : List<Vector3>, triangles : List<int>)

Figure 2.9: Class diagram of the `Graphics` component

shown in figure 2.11. The gold columns depicted are the outer columns described earlier. Also, the inner crystal can have various colors depending on the color of the laser beam required to hit it.

2.3.2 Mirror

A mirror is a crucial game element. Its reflective surfaces allow it to reflect any laser beam that hits these surfaces. It is also the only element that players can move and/or rotate. All levels require at least one mirror to move or rotate in order to hit the target. An image of a mirror in-game is shown in 2.12. The light blue circles reflect laser beams, the golden outer frame does not.

As mentioned before in the user activities, the mirror's outer frame will become brightly yellow when that mirror is selected. A selected mirror is shown in figure 2.13.

2.3.3 Wall

The wall is the main obstacle in the game. It blocks incoming laser beams completely. Walls are used in levels to make it harder for one player to reflect a laser beam coming from an emitter to the target. The first few levels mainly use walls to create paths that the laser beam has to go through, later levels use not only walls, but also other game objects. A wall is shown in 2.14.

2.3.4 Emitter

The emitter is the most important aspect of the entire game. It is the only "true" source of a laser beam (although game elements like the beam splitter

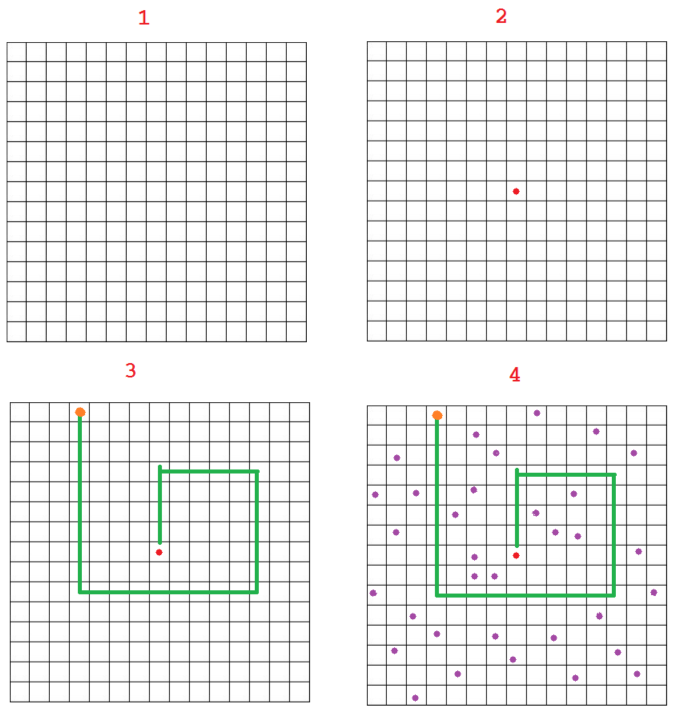


Figure 2.10: The random level algorithm, displayed graphically.

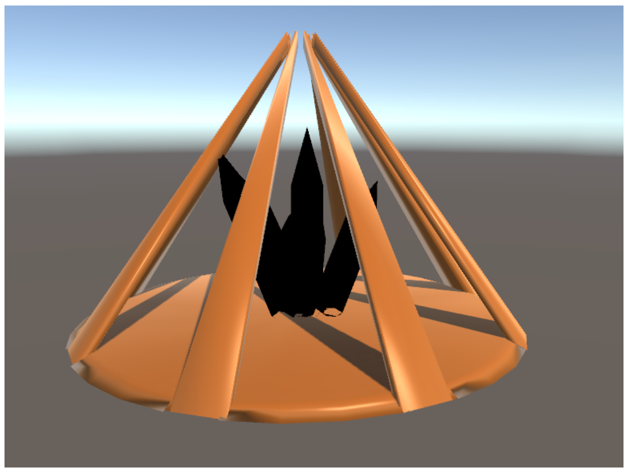


Figure 2.11: The laser target.

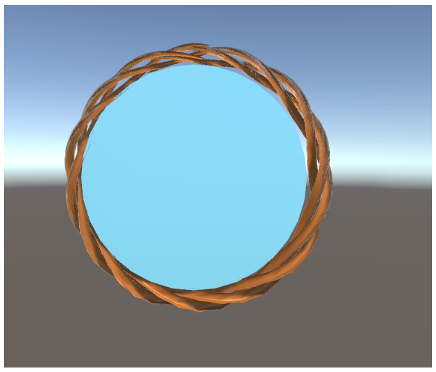


Figure 2.12: A mirror.

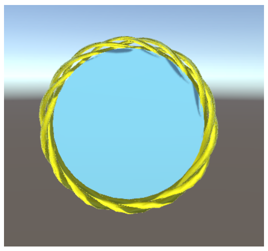


Figure 2.13: A selected mirror.

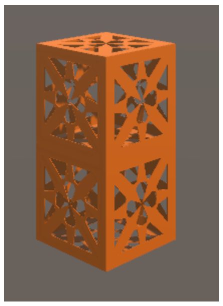


Figure 2.14: A wall.

can also create beams, these elements always require input in the form of another laser beam; the emitter does not have that problem, hence it is a ”true” source). In the early levels, players only move and rotate mirrors to guide a laser beam from the emitter to the target, while in the later levels beams have to be guided towards other game elements (like the beam splitter, for example) in order to complete the level. It is possible to have multiple emitters in a single level, and later levels use this to create more complex puzzles. What the emitter looks like exactly is shown in figure 2.15. Also shown here is a laser beam coming from the emitter. The laser beam is colored red per standard. It is also possible to alter the color of the beam coming from the laser, to be either blue or green.

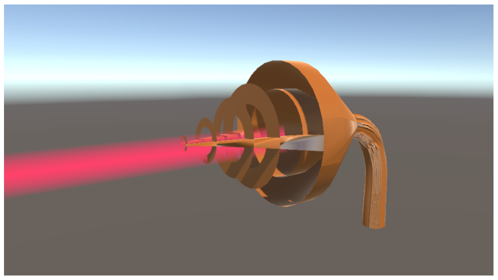


Figure 2.15: The emitter.

2.3.5 Checkpoint

The checkpoint is a very important game object. Once a checkpoint is hit, it is registered that the checkpoint has been hit until the laser beam hitting it is no longer aimed towards it. In order to proceed to the next level, one needs to hit all the checkpoints in the level as well as all the targets. Checkpoints do not require that the laser beam has any specific color in order for them to register that they have been hit. The checkpoints also allow lasers to pass through them. Checkpoints have been added mainly to prevent sequence breaking of a level (sequence breaking is a term used in video games for being able to get to a goal using a different route than you were supposed to, often making it far easier than it should be. Gamers making use of this ”break the sequence” of actions in the game, hence the term). An image of a checkpoint is shown in 2.16

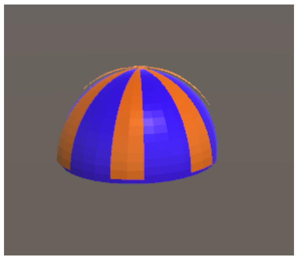


Figure 2.16: A checkpoint.

2.3.6 Portal

The portal is a more advanced game object. It allows light that travels into it to travel out of the portal it is linked to, and vice versa. This allows puzzles to contain targets that cannot be hit by just reflecting laser beams from the emitters by mirrors (as the laser beams have to travel through a portal in order to be able to hit the target), therefore forcing the players to redirect laser beams to portals. A portal is depicted in 2.17. It looks like a recolored mirror. It has one black side and one green side. The black side allows laser beams to travel through it to the portal linked to it. The green side blocks laser beams completely.

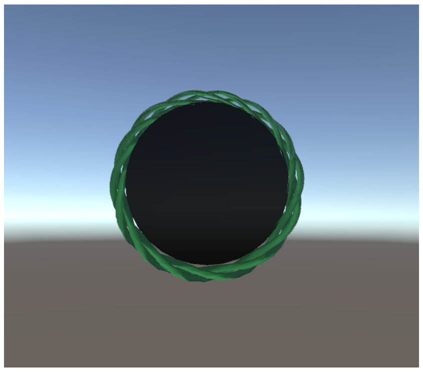


Figure 2.17: The portal.

2.3.7 Lens splitter

The lens splitter is another advanced game object. It splits incoming laser beams, creating two separate laser beams that each go a different way. This allows creation of another laser beam without needing an extra emitter, which allows hitting of multiple targets, for example. A lens splitter is depicted in 2.18. As can be seen, it has two lenses, to focus the laser beam to a specific point in the splitter; and a crystal, to split the incoming beam. The laser beam has to hit the outer lens in order to be split, and the outgoing beams will be emitted from the outer points of the crystal.

2.3.8 Planned game objects that didn't make it

There were some ideas for game objects that did not make the final cut. For example, there were plans to use AND- and OR-gates that received beams as input and emitted beams when enough input was given (this would be at least one beam for the OR-gate, and at least two for the AND-gate, while these gates

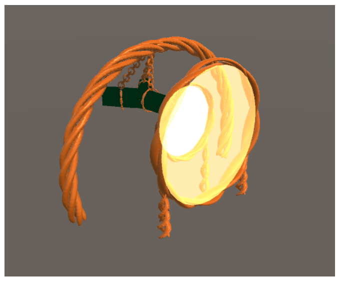


Figure 2.18: The lens splitter.

would then emit a single beam as output). These were the first logic gates that we thought about, and later ideas included XOR and NXOR gates (gates that would only emit a laser beam when hit by an odd or even amount of beams, respectively). There was (and still is) code that manages the behavior of AND- and OR-gates, however, there were no models, and after talking about them with our coach after the first demo, we concluded that these objects would not add much more to the game, and as such, development on them was scrapped.

Another idea that was eventually scrapped was the inclusion of switches that would be triggered by aiming a beam on them, which would then cause a wall linked to the switch to become transparent, allowing light to pass through it. This would cause players to collaborate to aim a beam to the switch, and then to aim another beam through the transparent wall. However, no models nor scripts were developed for this functionality, and the idea was eventually scrapped.

A final idea that was actually implemented and tested, but didn't meet the standard we set for it, was the elevator. This was a game object that used mirrors to elevate laser beams, to allow for passing over walls. The elevator was never used again, as it didn't work properly, and also because the portals overtook its function entirely.

2.4 Level design

Levels are designed to be simple at first, requiring little collaboration, while getting progressively harder, as well as introducing more and more game objects. The following sections describe the level categories as well as which game elements these levels introduce to the player.

2.4.1 Levels 1 to 5: the basic tutorial levels

The first five levels introduce the player to the most basic game elements that the game has to offer. These include walls, mirrors, and the laser target. Mechanics introduced and explained in these levels are: walls block incoming laser beams, mirrors reflect laser beams, one needs to hit all laser targets in order to proceed to the next level, and the color of the emitted laser beam should correspond to the color of the crystal it is hit with.

2.4.2 Levels 6 to 15: the easy levels

These levels serve to introduce the player to slightly more advanced game objects and their functions. As such, they introduce portals and lens splitters, as well as checkpoints. Game mechanics introduced in these levels are: portals allow a laser beam to travel instantaneously from A to B, lens splitters create two beams from one, and all checkpoints in a map need to be hit before one can proceed to the next level. The levels that introduce these features are designed to be impossible to complete without making use of these new features (this is an inherent function of the checkpoints, as one can only advance to the next level if all checkpoints have been hit, thus making use of the checkpoints). For example, when introducing the portal, the laser target is in a closed off area and can only be reached by using a portal to let the beam travel from A to B, with A being reachable by the laser beam on its own and B being in a closed off area, containing the target.

2.4.3 Levels 16 to 25: the advanced levels

These levels incorporate all the used game objects to create hard levels. The levels are a lot more difficult and require far more cooperation than the easy levels, from both the local and the remote players. No new game mechanics are introduced, but instead the cooperation skills of the players are tested.

Chapter 3

Implementation

The Bachelor Project course would not be a real Bachelor Project course if there were projects without technical challenges whatsoever regarding the implementation of the final result. Therefore, this chapter elaborates on the technical challenges faced during development of the project and the solutions that we have developed for these challenges.

3.1 AR Glasses

One of the first design choices that we had to make was about what AR glass was going to be used with the project. As can be seen in our research report, under appendix A, there two options to choose from. These were the Oculus and the META One. We eventually settled for the META One, because of the latency of the Oculus. A pair of META Glasses can be seen in figure 3.1.

The challenge with the META One was to get it working in Unity. There is a Meta SDK which allows for Unity games to work with the META One, but AR itself is still in development (as of writing this report), and the META One glasses are experimental at best. The SDK that we used first was also very buggy (due to the experimental nature of the META One). Also, the META One has a very limited field-of-view (the field-of-view was so limited that, during one of our first tests with our coach, the coach had to sit back to keep everything tracked, which, especially considering that they had to move markers as well as track the environment continuously, was less than ideal). However, the SLAM tracking built into the META allows for game objects to continue to be rendered on a marker even if the marker is outside of the view of the META. SLAM tracking is explained in subsection 3.1.1.

Figure 3.1: A pair of META One glasses.

3.1.1 SLAM tracking

SLAM is an acronym, which means Simultaneous Localization And Mapping. It stands for a computational problem making a map from an unknown environment while updating the location of the agent in the same environment. These problems cannot be solved independently from each other, as updating a map usually involves knowing the location of the agent before any accurate updates can be made, and vice versa. Several algorithms have been developed for solving this problem, and there is even a platform, called OpenSLAM, which contains several open source algorithms which solve this problem.

However, the algorithms are beside the point. the real benefit of using SLAM with AR is that SLAM tracking allows the META One to render the game objects belonging to a marker while keeping them rendered once the marker leaves the field-of-view of the META One. Considering that the field-of-view of the META One is not that large (As seen in our research report under Appendix A), this is a huge benefit. The META One also has built-in support for SLAM tracking of objects, which meant that no time had to be spent on developing algorithms.

3.2 Synchronization of World State

Because the game is played by multiple people, the state of the world somehow has to be synchronized between all players. To do this, we considered two major options:

The first option was to use the built-in Network View component in Unity. This would allow Unity to take care of most synchronization, which in turn could make implementing the synchronization particularly easy. However, due to the way the Meta One glasses manipulate the positions and rotations of game objects to fit the orientation of the player's head, synchronizing these positions would result in incorrect positions for other players. Instead, a custom serialization method would have to be implemented to undo the manipulation by the Meta One and then apply the correct manipulation for each of the other players.

Another option was to introduce a master server with camera that hosts the game, and provides raw positions and rotations of markers exactly as they were placed on the table. The only thing left to do would be to move and rotate the objects for each player to match that player's view of the playing area.

We chose the second option for the following reasons:

- A master camera can see all markers at any one time, which means that there is never any uncertainty.
- The first option requires complex peer-to-peer synchronization and conflict resolution when multiple players see an overlapping set of markers.
- Unlike the cameras worn by players, the master camera is not constantly moving, meaning that marker recognition is not affected by motion blur.

Aside from the aforementioned reasons, we also made the decision to go with the second approach because it allowed us more control over the internal workings of the network functionality and the marker tracking. See section 3.3 for the details about the marker detection performed by the server.

3.3 Marker Detection

The META One glasses come with marker detection built-in and we had difficulty replacing this detection with a custom system. That meant that our master camera server has to detect the same type of marker. Luckily the META markers have a very simple design. They're 6 by 6 bits encoded as black and white squares with a black border around them, as shown in figure 3.2.

These patterns map to an ID and are asymmetrical so that rotation can be resolved when they are detected. To easily find these markers on a table and board, we've also added a vibrant green border to them. This doesn't affect the META, but it makes segmentation of the markers from the background much more straightforward for the master camera. These final markers are shown in figure 3.3.

The corners of the playing surface were originally set using red markers, as shown in figure 3.4. The need for these corner markers is explained in the next sections, which also describe the rest of the marker tracking process on the server.

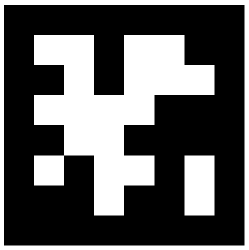


Figure 3.2: A marker that can be recognized by the Meta.

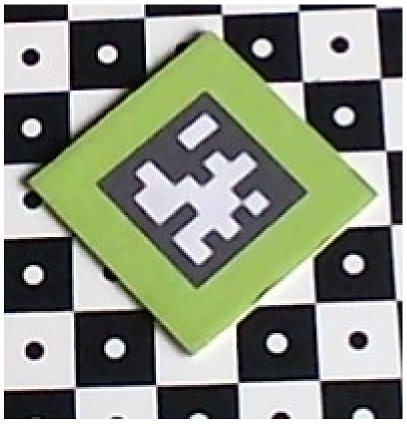


Figure 3.3: A marker that can be recognized by the Meta and the master camera.

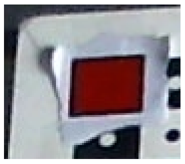


Figure 3.4: Original marker that indicates a playing surface corner.

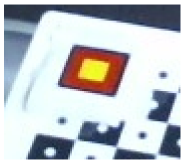


Figure 3.5: New marker that indicates a playing surface corner.

3.3.1 Board detection

The first step to tracking is to isolate the playing surface from the camera image and to apply perspective correction to it. The red corner markers described above are used to find the bounds of the rectangular playing surface. An example of the transformation is shown in 3.6.

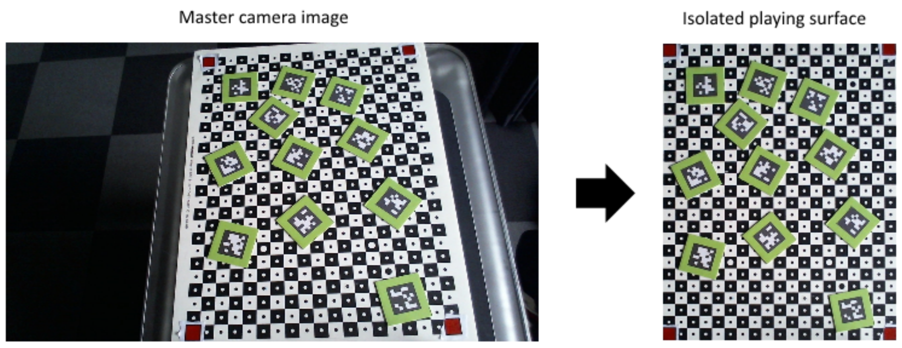


Figure 3.6: Example of detecting playing surface and isolating it.

The four red corner markers are located by converting the input image to the HSV color space and thresholding on a red-like hue and a high saturation. The noise is then removed using a morphological open operation and the system verifies that 4 contours are found. If an amount of contours other than 4 is found, the user is prompted to move the camera such that the entire playing surface is visible and there are no other vibrant red objects in view.

The corners of the playing surface are required to compute the transformation for the perspective correction. The perspective correction is required to recognize markers and their location correctly by making the camera view look like it views the board from above.

We had some issues with these solid red markers. For example, a red chair near the testing setup was often also recognized as a corner marker. For that reason we designed a slightly more complex corner marker that has a yellow center, as depicted in figure 3.5. Detection of this marker is the same as above along with a check for the yellow hole.

3.3.2 Marker detection

The markers are first segmented from the playing surface using their vibrant green borders, again through the HSV color space. The mask that results from this is cleaned up again using morphological open and close operations. An example of such a mask is shown in figure 3.7.

It is evident from the example that finding the borders alone is not sufficient, because they will connect when the markers are close together. However, we know that each marker contains a non-green center with the actual code, which shows up as holes in the mask. We can detect these holes by using OpenCV's

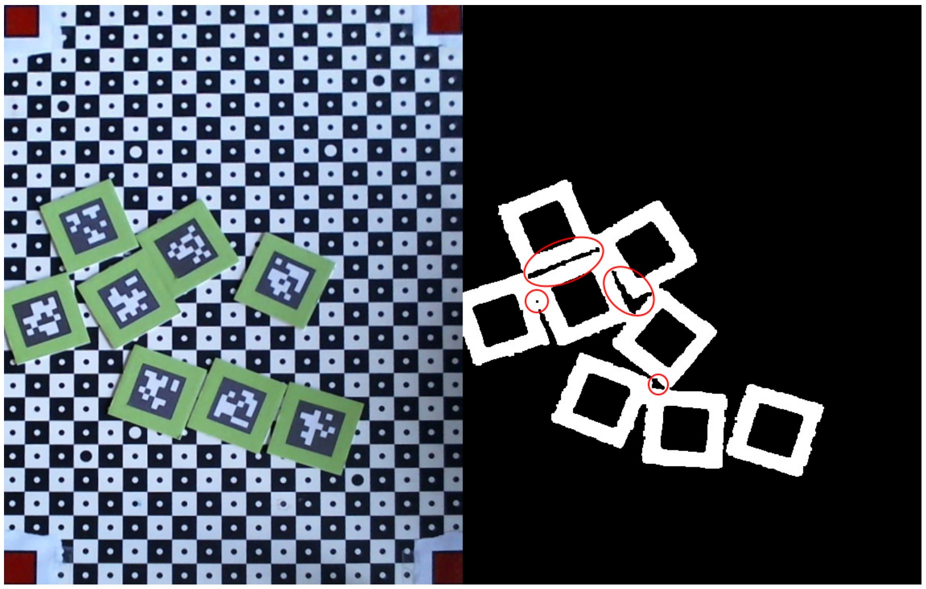


Figure 3.7: Example of thresholding markers with false holes highlighted.

contour finding feature using hierarchy mode and selecting just the contours that lie directly within the outer contours.

Unfortunately, when markers are really close together, some other holes appear as well (highlighted in the example). We remove these by filtering the contours using the following criteria:

- Holes have a minimum width and height of 8 (minimum required for reading a pattern)
- Holes are square to within a tolerance of 15%
- Holes have the same size as the median size of all detected holes to within a tolerance of 15%

Especially the last one works very well for a playing surface with many markers, where the signal to noise ratio is high. The process finished by returning the contours of all the holes detected as markers.

3.3.3 Marker recognition

The marker recognition process takes the contours from the marker detector stage. It starts by finding the minimum area bounding rectangle around a contour to find its rotation. The source image is then rotated to straighten the marker.

The next step is to isolate the pattern from the marker. It first converts the image to grayscale and resizes it to 8 by 8 pixels (the 6 by 6 pattern with the 1 pixel wide border within the marker). The average brightness across the pattern is found and used to threshold the black and white pixels. The image is then cropped to just the 6 by 6 pixel pattern. The original scale of the marker is also stored to later scale the marker positions.

The final step is to identify which known pattern the detected pattern best matches. Although the marker has already been straightened, it could still be rotated by 90, 180 or 270 degrees. For that reason, the system searches all known patterns using the four possible rotations of the input. It determines the best match using the Hamming distance between 2 patterns. The detected best matching rotation is then added to the rotation needed to straighten it to compute the complete rotation.

The marker recognition system does not verify if the best pattern match is good enough, it will happily return a best match with confidence score `0.0`.

3.3.4 Marker tracking

The marker tracker system takes the output from the marker detection and recognition stages (position and match) from each frame and uses this data to track the movement of markers across frames. It primarily does this by checking which marker position from the previous frame each new marker position is closest to. It uses the pattern matching results only when a marker is not moving quickly. It uses results from multiple frames to smoothen positions and rotations using a moving average that is discarded when a marker has sudden changes. It also detects if a marker has been removed by measuring if it hasn't been seen for a long while.

All of the changes it detects, like new markers, moved markers and removed markers are added to a list each frame and returned to the main application. The main application then broadcasts these changes over the server socket to the META One clients.

3.4 Networking and the OpenCV server

The game depends on a server application with a master camera. The server application detects the position and rotation of markers in the playing area. It does this through the use of a central so-called master camera, that is position so that the entire playing area is visible from the camera.

The server application is written in C++ and is based on OpenCV and the Qt framework. We decided to implement the server outside of Unity, since Unity does much more than what we need of the server. The server only acts as a way to track all markers even if they aren't seen by any of the players, and to facilitate synchronizing state changes with all players. For example, if a remote player rotates an object in the game, the details about that rotation is sent to the server, which then distributes it to all other players.

A more detailed description about the communication between the Unity clients and the server can be found in paragraph 3.4.1. The use of the master camera to detect markers is described in paragraph 3.4.2.

3.4.1 Communication between C# and C++

Communication between the Unity clients and the OpenCV server happens through the use of sockets. For the Unity side, the Socket facility built into the C# runtime is used. For the OpenCV server, the TCP Socket facility of the Qt Network module is used for providing a server socket capable of handling multiple clients.

The protocol used for communication is kept very simple to reduce network load and for simplicity. The protocol consists of a number tag indicating the message type followed by the actual message content. To facilitate the features the game provides, the following message types are used:

Position Update Sent by the server whenever it detects a change in a marker position.

Position Delete Sent by the server whenever a marker is removed from the playing field.

Rotation Update Sent by remote players to indicate they have rotated an object. This message is forwarded to all connected clients by the server.

Level Update Sent by players as soon as they consider the level to be finished. The server broadcasts it to the other clients to change the server.

Ping Message Sent by the server and clients to indicate they are still connected and listening.

3.4.2 The master camera

The master camera uses the tracking system described in section 3.3. This system outputs the positions of the markers in pixels and rotations relative to the camera. The positions are normalized such that the width and height of a marker pattern is 1 unit. The detected scale from the marker recognition module is used to do this and is averaged across all markers and a couple of frames to ensure stability.

The normalized positions and relative rotations are sent to the META clients using the communication protocol discussed in the previous section. Once they arrive, they are transformed into the META coordinate system using a ”level marker”. A ”level marker” is an arbitrary marker detected by both the master camera and META built-in tracking system. The client can then transform the relative position of the other markers compared to the level marker from the server coordinate system to the META coordinate system. The rotation of the markers is compensated using the META detected rotation and the rotation received from the server.

Chapter 4

Quality Assurance

This chapter explains and describes various quality assurance techniques that were used during the project, to allow us to deliver a product of good quality (regarding both code quality and gameplay quality). It also talks a bit about Microsoft Visual Studio first, our editor of choice for this project, considering that some QA tools are not available for MonoDevelop, the standard editor packaged with Unity.

4.1 IDE used for programming

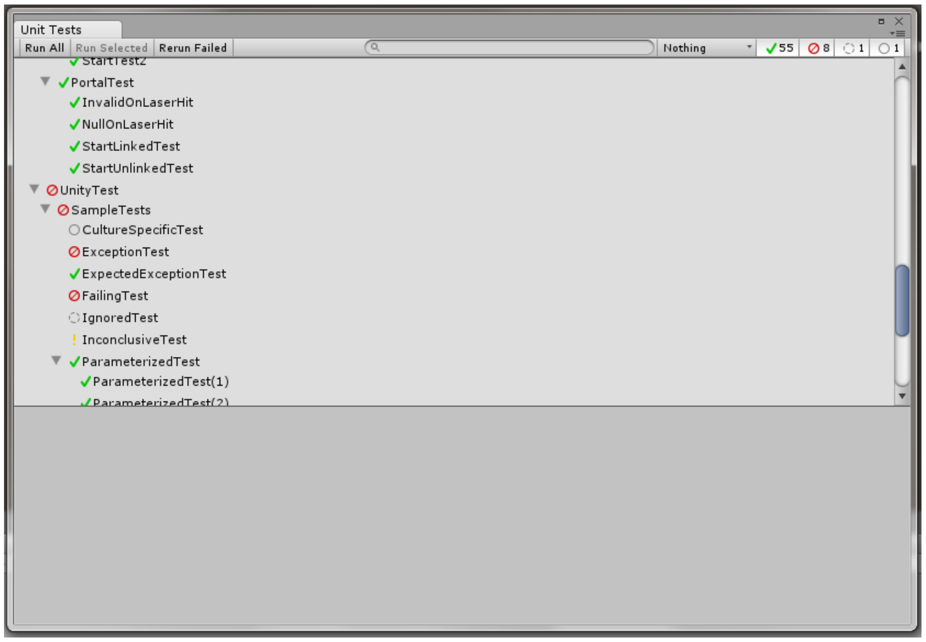
The IDE used for programming was Microsoft Visual Studio. Unity has an editor of its own available for programming in C#, which is called MonoDevelop. However, the MonoDevelop IDE is really lacking in functionality. It has no support for the plugins that we use to check our code. Furthermore, the use of MonoDevelop enforces a code style that is incompatible with the style guidelines used by StyleCop (see 4.3),

4.2 Testing

There are three main types of testing done during the project. These are unit testing, integration testing and user testing. Unity has no native support for running unit/integration tests that have been written, but there is a toolkit available for free on the Asset Store, called Unity Test Tools, that does have this support. The extension is developed by the Unity team, and can be found on the Unity Asset Store: `https://www.assetstore.unity3d.com/en/#!/content/13802`. Using this extension, a new menu bar item, called "Unity Test Tools" will appear in the main Unity editor. Clicking on this item creates a drop down menu with different options, the most important one being the unit test runner.

4.2.1 C# Unit Tests

Unit tests are written using the NUnit unit testing framework for C#. NUnit is a test framework which was ported from the Java test framework JUnit, and was created to bring xUnit testing to all .NET languages. Using this framework is also really easy, and a tutorial on how to write unit tests using NUnit can be found using Google. Using Unity Test Tools, all unit tests in the project are listed once one clicks on the subitem ”Unit Test Runner”. The tests are listed in a new window, and one can run all unit tests by clicking on the ”Run All” button at the top. The menu then shows what unit tests have passed or failed, and clicking on a unit test shows what went wrong. A unit testing overview can be seen in 4.1. The UnityTest testing class seen in the overview also displays the different statuses of tests in the NUnit framework (passing, failing, inconclusive, not executed, and culture specific).



```
      ✓StartTest2
  ▼ ✓PortalTest
      ✓InvalidOnLaserHit
      ✓NullOnLaserHit
      ✓StartLinkedTest
      ✓StartUnlinkedTest
▼ ⊘UnityTest
  ▼ ⊘SampleTests
      ○CultureSpecificTest
      ⊘ExceptionTest
      ✓ExpectedExceptionTest
      ⊘FailingTest
      ◌IgnoredTest
      !InconclusiveTest
    ▼ ✓ParameterizedTest
        ✓ParameterizedTest(1)
        ✓ParameterizedTest(2)
```

Figure 4.1: The Unity Test Tools unit testing screen.

4.2.2 C++ Unit Tests

The unit tests for the C++ server code are not written using the NUnit framework, as that would not really work. The framework for the tests in C++ is the Google Test framework. Google test is, like NUnit for C# and JUnit for Java, an xUnit-based testing framework. As such, it also supports assertions, type parameterization, etc. Also, it is open source, licensed under the new BSD license.

To implement the testing functionality into the server project, the unit tests are separated into their own subproject, which links to the server and runs the tests. As recommended by the Google Test guide, the Google test framework source and headers are included directly into the unit tests subproject, ensuring that the tests can be compiled and run regardless of the platform or compiler being used.

Because most of the functionality implemented in C++ involves computer vision, we've prepared images so that situations can be reliably reproduced for test cases. This allows us to test the algorithms under many different lighting conditions at once, for instance, which saves us a huge amount of time. Some of the tests involve functionality that returns an image as result. We've created the image we expect and have written utility functions to check if two images are approximately equal.

4.2.3 Integration Tests

Integration tests are not done via a formalized test procedure, but rather by creating simple scenes and observing that the subjects of the test work as they should when they are placed in an actual scene. It is also a lot harder to run these tests in a standardized way most of the time.

4.2.4 User Tests

Although testing the code is important and it helps ensure that the software is working correctly, it doesn't tell us anything about the actual usability of the product. Since we have been developing a game, it is especially important that the end users will have fun using the product. Properties like these can be evaluated by doing user tests.

Most of the initial play testing has been done by us, the developers, because external parties would not have had a good experience while the game was still in the middle of development. However, as the project was nearing completion, the game could no longer be objectively played and tested by the developers. At that point, we were already too familiar with the levels and game mechanics to be able to properly judge if the concepts are too hard to figure out for new players. By having a lot of other people play the game, we were able to estimate if certain levels were either too easy or too hard, and if all of the different game objects were easy to understand and fun to interact with.

Finding other computer science students to play the game was not much of a problem, but we also needed to have other people play the game, to get a good overview of how different people perceive the game. This is especially important, as computer science students mostly indicated the technical issues and limitations of the Meta One glasses, which were mostly already known to us.

4.2.5 Code coverage

Code coverage is a metric that can be used to determine how thorough the written code has been tested. We have to consider generating code coverage reports for two languages: The C# Unity project and the C++ server project.

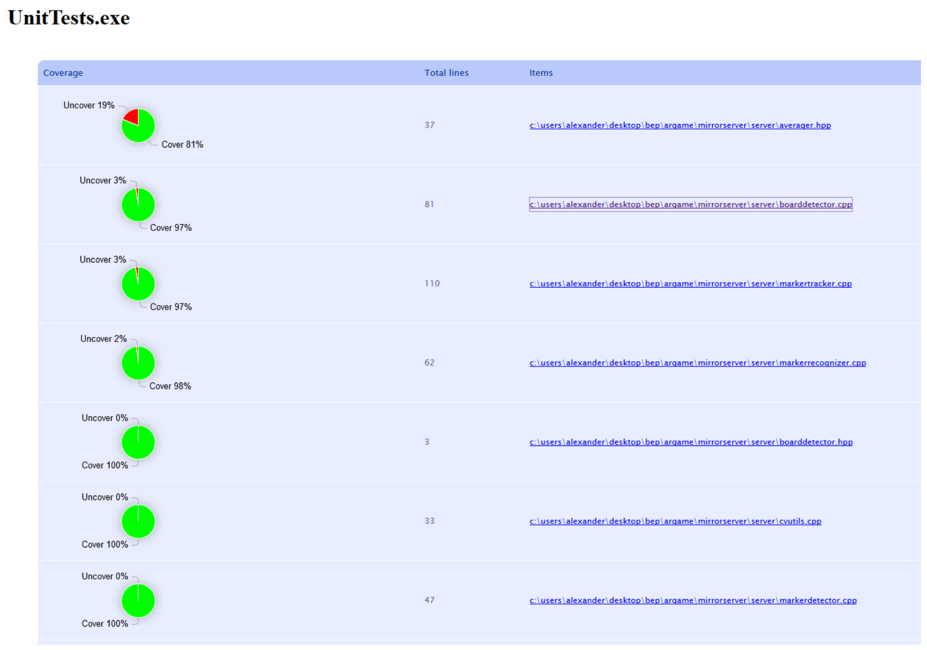
Generating code coverage reports for the C# code is unexpectedly difficuly: Even though here are several free software packages available on the Internet that allow for generating code coverage reports of NUnit test suites, these are hard to use when combined with unit tests in Unity. The reason behind this is that, to run the unit tests for the game, the Unity Test Tools functionality has to be used (as most tests use instantiation of gameplay objects, something that canonly happen in Unity). This functionality has no way of integrating the NCover or OpenCover software packages. It is possible to integrate these with Microsoft Visual Studio, however it is impossible to run the unit tests from that IDE. As such, we had to manually check if the tests tested all possible branches of the code.

On the other hand, however, analysing code coverage for the C++ project is almost trivial: For GNU compilers, including MinGW, there is a utility called gcov, which measures code coverage of GTest-based unit tests. And when using the Microsoft Visual C++ compiler, it is possible to use Visual Studio to analyse the coverage of the unit tests. The tool used for analyzing C++ code coverage through tests is called OpenCppCoverage, which is compatible with Microsoft Visual Studio 2008 and later. It easily measures unit test code coverage, and also does not require any extra tools to actually generate the report that describes the measuring results. These are immediately afterwards given in a HTML file. An example of such a report and its results is given in 4.2, while an example of which lines of code have been analyzed is given in 4.3.

4.3 Code Style

We decided to stick to the code style guidelines defined by StyleCop and FxCop, two utilities developed by Microsoft. These utilities check for common programming and code style errors in projects so that these can easily be identified and fixed.

During the project, we kept the source code style checked by periodically dedicating time solely for checking code style and performing code maintenance. This also included writing or improving unit tests and refactoring classes and methods with a relatively high complexity or other issues as indicated by StyleCop and FxCop.



UnitTests.exe

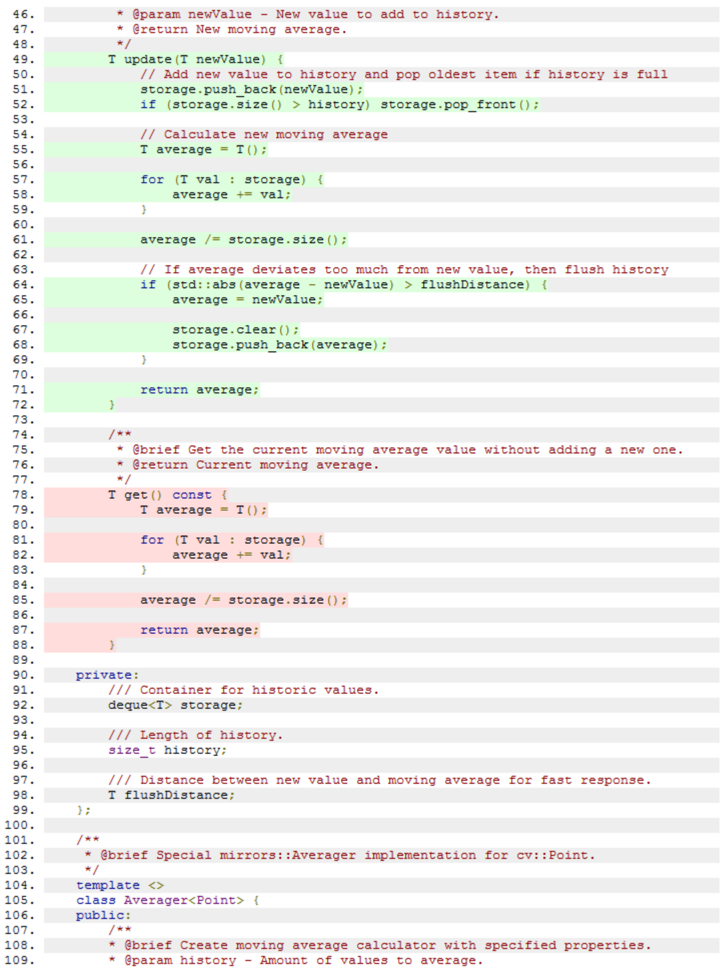
Coverage	Total lines	Items
Uncover 19% Cover 81%	37	c:\users\alexander\desktop\bep\argame\mirrorserver\server\averager.hpp
Uncover 3% Cover 97%	81	c:\users\alexander\desktop\bep\argame\mirrorserver\server\boarddetector.cpp
Uncover 3% Cover 97%	110	c:\users\alexander\desktop\bep\argame\mirrorserver\server\markertracker.cpp
Uncover 2% Cover 98%	62	c:\users\alexander\desktop\bep\argame\mirrorserver\server\markerrecognizer.cpp
Uncover 0% Cover 100%	3	c:\users\alexander\desktop\bep\argame\mirrorserver\server\boarddetector.hpp
Uncover 0% Cover 100%	33	c:\users\alexander\desktop\bep\argame\mirrorserver\server\cvutils.cpp
Uncover 0% Cover 100%	47	c:\users\alexander\desktop\bep\argame\mirrorserver\server\markerdetector.cpp

Figure 4.2: The C++ code coverage report.

4.4 SonarQube

For getting a clear overview of the source code quality of the project, as well as the issues indicated by FxCop and StyleCop (see section 4.3), we made use of a SonarQube server, hosted on one of our development machines. SonarQube keeps track of issues indicated by the abovementioned tools, and performs various code metrics, like cyclomatic complexity and dependency cycles. Using SonarQube enabled us to spot problematic classes and methods and allowed us to improve the overall structure of the source code of the project. An example of a SonarQube overview can be seen in figure 4.4

An issue with SonarQube is that, while it has free options for analyzing C# code, it has none of these free options for analyzing C/C++ code, because of the preprocessing that can happen in C or C++ (as explained in their blog on `http://www.sonarqube.org/ccobjective-c-dark-past-bright-future/`). For this reason, it is very hard to analyze the C++ code that we use for the OpenCV server.



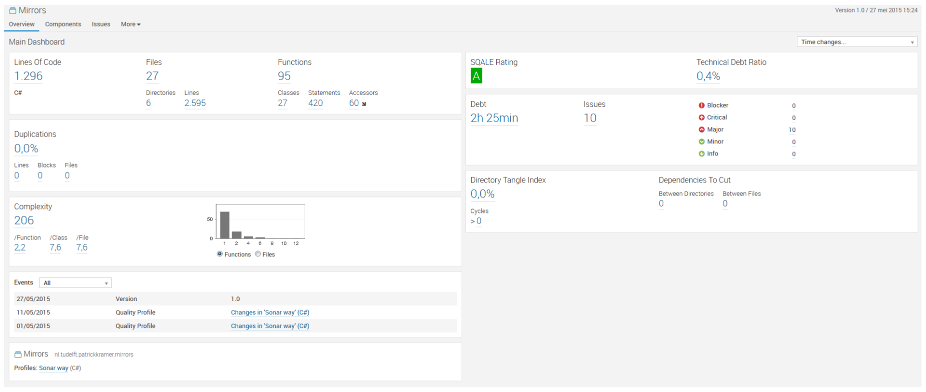
```
         * @param newValue - New value to add to history.
         * @return New moving average.
         */
        T update(T newValue) {
            // Add new value to history and pop oldest item if history is full
            storage.push_back(newValue);
            if (storage.size() > history) storage.pop_front();

            // Calculate new moving average
            T average = T();

            for (T val : storage) {
                average += val;
            }

            average /= storage.size();

            // If average deviates too much from new value, then flush history
            if (std::abs(average - newValue) > flushDistance) {
                average = newValue;

                storage.clear();
                storage.push_back(average);
            }

            return average;
        }

        /**
         * @brief Get the current moving average value without adding a new one.
         * @return Current moving average.
         */
        T get() const {
            T average = T();

            for (T val : storage) {
                average += val;
            }

            average /= storage.size();

            return average;
        }

    private:
        /// Container for historic values.
        deque<T> storage;

        /// Length of history.
        size_t history;

        /// Distance between new value and moving average for fast response.
        T flushDistance;
    };

    /**
     * @brief Special mirrors::Averager implementation for cv::Point.
     */
    template <>
    class Averager<Point> {
    public:
        /**
        * @brief Create moving average calculator with specified properties.
        * @param history - Amount of values to average.
```

Figure 4.3: The C++ code coverage report, with covered and uncovered lines.

4.5 SIG Evaluation

SIG is an acronym which stands for the Software Improvement Group, which is a company that is based in Amsterdam. SIG performs code analysis to evaluate code quality on a scale from one to five stars, and it does so according to the ISO/IEC 25010 model. Code score is based on the maintainability of the code.

During the project, there are two opportunities to deliver the code that we have written to SIG. The first opportunity is at the end of May, and the second is halfway through June. The first opportunity is used as a midterm quality feedback session, for us to get an idea about how good the code is written and what can be done better. The second opportunity is then intended to hand in



Version 1.0 / 27 mei 2015 15:24

Main Dashboard

Lines Of Code 1.296
C#

Files 27
Directories 6 Lines 2.595

Functions 95
Classes 27 Statements 420 Accessors 60

Duplications 0,0%
Lines 0 Blocks 0 Files 0

Complexity 206
/Function 2,2 /Class 7,6 /File 7,6

Events All

27/05/2015	Version	1.0
11/05/2015	Quality Profile	Changes in 'Sonar way' (C#)
01/05/2015	Quality Profile	Changes in 'Sonar way' (C#)

Mirrors nl.tudelft.patrickkramer.mirrors
Profiles: Sonar way (C#)

SQALE Rating A

Technical Debt Ratio 0,4%

Debt 2h 25min

Issues 10

Blocker	0
Critical	0
Major	10
Minor	0
Info	0

Directory Tangle Index 0,0%
Cycles > 0

Dependencies To Cut
Between Directories 0 Between Files 0

Figure 4.4: A SonarQube overview.

the improved code, and for us to then get feedback on how well the code has been improved. The improvements in the code also have a weight in the final result for the project.

The midterm and final evaluation reports we received from SIG are added as appendices C and D, respectively.

4.6 Demos and playtesting sessions

In order to see how other people would interact with our software (and also partially to see if solving the game actually required collaboration of multiple people), several demo's with the client were hosted. Because of time constraints, only one play testing session was hosted before the final report had to be handed in. The main output of that play testing session was that the technology was far from perfect (which is something that we knew from the start, so that was not really useful). However, these test subjects were other EEMCS students, and there were plans to test with students of other faculties, but that was cut due to time constraints (as mentioned above).

The demos with Stephan (the client) hit a rough start the first few times, as the game that would be demonstrated would often not work right before the demo was held. After some demos, however, the game was far enough into development that it would be easy to prepare an actually working version. The best demos were demos hosted after week 9 (right before the project had to be finished), because the game was nearly done. The feedback of these demos is that Stephan liked the game, and that he looked forward to playing it. This means that the client likes our product, a positive outcome.

Chapter 5

Conclusion

Despite some rocky development, we ended up with a fully playable game in week 9 that met all of the must have and almost all of the should have requirements. The should have requirements included mirror elevations, but we found that this wouldn't work well in practice and replaced this feature with portals. Those accomplish the same purpose of transferring a laser beam across a wall. We've also done away with color combiners because the results of mixing colors would be too complicated for players. The game can support this, however, and these color mixers would be easy to implement. The main takeaway here is that some of the should haves were not met for gameplay reasons rather than technical problems.

The only component of the game that is still rather lacking is the one that gave us the most trouble during development: marker tracking with AR glasses. The glasses we are using for the game (Meta One) represent the state-of-the-art of augmented reality see-through glasses, but they have a field-of-view of just 35 degrees and a marker tracking framerate of about 5. The first issue causes players to have a lack of overview unless they take quite a few steps back from the table. The second issue causes the mirrors to lag behind when the player moves their view, which diminishes the immersion of the game. We repeatedly contacted the company behind the eyewear, but there were no plans to improve on these issues any time soon.

Despite these limitations of the augmented reality technology, we have been able to produce a fun and complete game to play that can be used to experiment with situational awareness like the client desired.

Appendix A

Research Report

A.1 Problem Formulation

While augmented reality research has grown into a mature field over the last years, the aspects of situational awareness and presence of augmented reality (AR) are still quite open research topics. This project is about designing and implementing a collaborative game to explore the different perception of situational awareness, presence and workload in a physical and an AR environment. The game is to be employed as an approximation of collaboratively solving complex problems, as they occur in crime scene investigation when using virtual co-location, i.e. expert remote crime scene investigators to guide local investigators in AR to collaboratively analyse the crime scene.

The game needs to support at least three players: At least two players are present at the same location (physically co-located), each wearing AR glasses. At least one player is physically remote but virtually co-located using VR glasses [Lukosh, 2015]. It should be impossible, or at least infeasible, to complete the game without involving the other party. However, this constraint could be relaxed to allow a higher playability of the game. It would be nice to still be able to play the game if no suitable virtually co-located player can be found, for example.

A.2 Problem Analysis

This chapter provides an overview of the issues and challenges that may arise during development of a solution to the problem description. It provides an analysis of the problems and possible solutions.

One of the core challenges of the project is the use of Augmented Reality (AR) technology. An analysis of the available options to implement this functionality is given in section A.2.1. Another important challenge is improving situational awareness, which is discussed in section A.2.2. The final challenge is the creation of interdependence between players in such a way that collaboration from all players is required. This challenge is analyzed in section A.2.3.

A.2.1 Augmented Reality (AR) Functionality

Augmented Reality (AR) is a core aspect of the problem formulation. As such, careful analysis has to be done as to how the AR functionality can be best implemented to fully address the context of this project.

We consider three choices for implementing AR functionality: The META One (an optical see-through device (A.2.1)), the Oculus Rift virtual reality glasses in conjunction with mounted cameras (A.2.1) and a smartphone with Google Cardboard (A.2.1).

META One

The META One glasses are optical see-through glasses. [Meta, 2015] Optical see-through glasses work by projecting a virtual image on top of the world you see, effectively implementing a 3D AR exprience.

Because the META One is an optical see-through device that also features motion tracking, AR can be implemented simply by projecting an image against a black background to the glasses.

A big drawback of the available META One glasses is their Field-of-View, which is 35 degrees. This Field-of-View is way lower than the Field-of-View of a person, which may have a negative impact on the game experience.

One of the advantages of this device is that the Software Development Kit (SDK) that comes with the glasses has tracking and gesture recognition built-in. That allows us to focus on just the gameplay aspects.

Oculus Rift

We've built a camera rig for the Oculus Rift that can be used to turn it into an augmented reality device. To detect the markers and render objects on them in Unity, there are several libraries available. Each of these will be discussed in the next sections.

Oculus offers an SDK for Unity that makes it easy to integrate a game with the Rift. [OculusVR, 2015] The challenge that we'll be facing during development is to properly integrate this SDK with the augmented reality libraries. Each of the frameworks try to take control of the camera in different ways and it's easy to get conflicts there. Getting the Rift see-through functionality working in Unity on its own and the augmented reality functionality on its own is not a challenge.

The advantage of the Oculus Rift with mounted cameras is that it provides complete control over the screen allowing us to completely block certain regions of the view or project black objects. It also causes the AR content to match up exactly with the real world in terms of timing. The field of view is significantly larger as well.

A disadvantage is the slight latency between head movements and the view, which takes work to reduce to a minimum to avoid motion sickness and bad hand-eye coordination.

Vuforia

Vuforia is a framework by Qualcomm that allows you to create arbitrary markers, import them into Unity and place objects onto them. [Qualcomm, 2015] You can then select a webcam and have it render the camera images with 3D objects projected onto the markers. It's very easy to use and has built-in support for virtual reality solutions like GearVR. The tracking quality is very good and stable, even with low quality markers (with few color transitions).

Unfortunately it currently only works with the 32-bit version of Unity. It also lacks support for the Oculus Rift on the desktop, which means that we'll have to build that functionality ourselves.

Unity AR Toolkit (UART)

Unity AR Toolkit [MacIntyre, 2012] is a project by researchers at the Georgia Institute of Technology to develop a set of plugins for Unity that make it easy to build augmented reality applications.

Unfortunately the project was abandoned in 2011 and the tracking library it's based on (ARToolKitPlus) was abandoned in 2006. The documentation is also severely lacking. The camera also needs to be calibrated to work with it. This project also only works with the 32-bit version of Unity and lacks native Rift support.

The one advantage of this library is that it moves the 3D objects by default, instead of the camera. That may make it easier to reconstruct the world with its mirrors in Unity.

Metaio

Metaio is a well-supported project, much like Vuforia. The difference is that it has native support for see-through stereo glasses, which makes it well suitable for both the META One and Oculus Rift solution. [Metaio, 2015]

Unlike any of the other solutions, this framework works natively on Unity 64-bit. It does have the limit of only working with OpenGL, but we don't expect that to be a problem.

Google Cardboard

Google Cardboard can be used to implement the same concept for augmented reality as the Oculus Rift. It allows you to turn a smartphone into a VR headset by mounting lenses in front of it. This can then be used to implement AR by overlaying 3D objects over the phone camera image. [Google, 2015]

The advantage of this approach over the Oculus Rift is that it's much more convenient. The solution would be completely mobile, meaning that there are no cables getting in the way (a problem we ran into during testing). It's also much cheaper, making it easier for local players to join by simply owning a smartphone. The disadvantage is that there would be no depth, since most smartphones don't have a stereo camera.

The libraries mentioned in the previous section are all suitable for mobile usage. In fact, they were designed for it and just happen to allow for desktop usage as well. This currently seems like the most promising alternative, but it will require further testing.

Markers

An essential component of augmented reality systems are markers. Markers are physical objects that are used by the augmented reality framework to establish the position of the player and objects in the virtual world (tracking). There are two main types of tracking:

- **Marker tracking:** Special asymmetric patterns similar to QR codes are printed to cards. These are designed to be efficiently detected by a computer
- **Markerless tracking:** Tracking with arbitrary images as opposed to specially designed markers, therefore known as markerless tracking. This name is a bit misleading, because these images are still typically chosen based on properties like contrasting colors and sharp edges that are easier to detect.

The advantage of markerless tracking is that they look more aesthetically pleasing to the players, but the performance may be worse than specially designed patterns. For that reason, we're going to use a hybrid form that combines an easy to scan pattern with a nice looking image of the object it represents (like a mirror). An example of this concept are graphical QR codes. [Visualead, 2015]

A.2.2 Situational Awareness

This project is about exploring the different perception of situational awareness, presence and workload in a physical and an AR environment (see chapter A.1). As such, situational awareness plays a key role in this project.

Before considering exactly how situational awareness plays a role in this project, it is important to define precisely what situational awareness means. According to [Endsley et al., 2003], situational awareness is defined as *the perception of the elements in the environment within a volume of time and space, comprehension of their meaning, and the projection of their status in the near future.* In other words, situational awareness means to fully understand the situation, and be able to predict what is going to happen next. This also includes understanding any risks the situation brings.

Now that the concept of situational awareness has been defined, the importance of situational awareness for this project needs to be considered. This project aims to approximate a situation in which the players need to collaboratively solve complex problems, as they occur in crime scene investigations (see chapter A.1). On a crime scene, the investigator needs to analyse and understand the situation, so there is a need for situational awareness in the problem description. The game will need to replicate a similar need for situational awareness to be an accurate approximation of the context of this project.

A.2.3 Interdependence between players

The problem formulation states that the game is to be employed as an approximation of collaboratively solving complex problems. In order to motivate players of the game to collaborate, there is a need to create a form of interdependence amongst the players, as mentioned in [Zagal et al., 2006]. One way to do this is to create an asymmetry of abilities. Other ways to create interdependence are explained in the following subsections.

Asymmetry of abilities

The main reason to co-operate is the asymmetry of abilities between the players involved. For example: physically co-located players can alter the game world, while virtually co-located players can guide characters to a certain goal utilizing the altered game environment. One thing to note is that a *puppet master* scenario should be avoided. This scenario happens when one player can do everything except for a few required tasks, and uses the other players to execute these tasks. In this case, the other players will have less involvement with the shared goal, and the amount of co-operation will go down.

Asymmetry of information

Asymmetry of information could be used as another reason for the players to co-operate. It means that both types of players (both the physically co-located and the virtually co-located) have different, separate, parts of the information required to complete the game. In this case, a *puppet master* scenario should also be avoided. Such a scenario can occur here when one type of player has enough information to infer nearly all information.

Information overload

Another reason for players to co-operate could be information overload. This means that, while the game could technically be completed by a single player, the amount of incoming information is too large for one player to handle. An example would be in Call of Duty, where people work together because there are too many enemies that walk around and shoot at them. A single player could potentially beat the game by themselves, but this is not really feasible considering the amount of enemies and the amount of information coming in continuously.

A.2.4 Virtual Co-location

Establishing virtual co-location is required to allow physically remote players to play the game together. As such, both the virtualization of the game world and the networking are considered in virtually co-locating physically remote players. Unity has multi player support, because of its master server to handle multi player games, but the server could be down at times. There are tutorials on the internet to create a basic multi player game that uses the master server to handle requests. These tutorials can be used to implement our own multilayer support. Alternatively we could provide the players with the means to easily get and exchange their IP addresses through other means such as mail. Besides the networking we have to look at how we synchronize locations, depending on the chosen game we have in order of ascending complexity several options:

1. Use markers with known locations. This only works if we have a limited size and reasonably fixed playing area which we can prepare ahead of time. This most likely comes in the form of a set of markers on the edges and/or center of the playing field.
2. Use mobile markers which synchronize between players automatically, for example cards in a card game. This only works if we can trust the player to keep these markers within their screen or if there is no augmentation needed if they cannot see a marker.

3. Object recognition which tracks the locations of objects in the scene, this method only works if there are a number of reasonably stable objects within the player's vision.
4. Combining the output of a compass, a gyroscope and trilocations. This works regardless of what is visible but requires accurate trilocation which works can be quite hard to do without building a heavy rig.

Of course a combination of several of the above methods is also possible.

In the end the goal of the project is of course to make the remote player feel as if he is in the same location as the local player and make the local player feel as if the remote player is playing together with them. So first of all we can look at the visualization for the remote player:

One method is to provide an Oculus Rift to the remote player(s) and let them see through the eyes of the local player(s). However, this is likely to cause nausea. Of course we can display the local player's view on a screen instead, but that might result in relative passivism from the remote players, as they do not have any control over their own view.

We can also let the remote player control one or more avatars within the game world and view these through either the Oculus Rift or through a screen. This would keep the remote player more interested as it would add a greater feeling of immersion than just watching through the eyes of the other player. However, this comes at the cost of the feeling of connectivity. This would also require mapping out a large part of the scene in the virtual world.

Lastly we can also let the remote player view the world from a bird's-eye perspective. This can either be done by mounting a camera above the scene or by rendering it in the virtual world. The second case offers increased player agency resulting in improved situational awareness at the cost of having to map the scene fully in the virtual world.

We should also look at how we can make the local player feel as if the remote player is playing together with him. One method is to heavily encourage communication. If you are talking with someone it is hard to forget their existence. This can be encouraged by providing appropriate communication channels such as voice or text chat. Another important thing is that the remote player must visibly be doing something. This can be achieved by giving him an avatar and by making his actions visibly change the world.

A.3 Proposed Solutions

A.3.1 Laser mirror game

The goal of the game is to solve a puzzle by controlling laser beams using mirrors in such a way that a predefined target is hit. The game can be played by one or more local players and one or more remote players.

There are cards present for the local players that represent mirror bases. These must be placed on the table, which will be the locations for the mirrors. The local players will be able to see the mirrors they place through the use of AR technology. Each of the local players will only be given a few of the mirror bases needed to solve the puzzle, and as such solving the puzzle requires cooperation from all local players.

The remote players can also see the placed mirrors, and can rotate them to influence the path of the laser beam(s). Only by cooperation between local players (who can only move the mirror bases) and remote players (who can only rotate them) it becomes possible to hit the target and as such solve the puzzle.

The game provides various different types of mirrors with different properties, allowing for more complex puzzles. One example of such a mirror is a colored mirror, and then require the target is hit with the right (combination of) colors. Another way to make puzzles more complex is requiring that the players combine beams together to create more powerful beams. Other optical components like beam splitters can also be introduced.

The game is designed to stimulate cooperation between the physically co-located players and the physically remote player(s). It does so by dividing abilities required for solving the puzzles amongst all players as follows:

- Physically co-located players each get only a part of the mirror bases required to solve the puzzle, requiring input from all of these players.
- If there are multiple virtually co-located players, each of these players can only rotate a subset of the mirrors, and as such input from all virtually co-located players is required for solving the puzzle.
- Virtually co-located players have the ability to rotate mirrors while the physically co-located players do not have this ability, requiring input from both physically as well as virtually co-located players.

Because of this division of abilities, there is an interdependence between all players (see section A.2.3), regardless of whether these players are physically co-located or not. This replicates the interdependence that exists in the crime scene example as given in the problem formulation (see chapter A.1).

A.3.2 Platformer

The game starts with the avatar of the remote player appearing on the table and a goal appearing above the table. The local players have a pile of blocks, each of these blocks exists both in the virtual and real world.

Inside the virtual world a number of pre-existing structures and obstacles exist making it harder for the remote player to move around. The local and remote player must work together in order to get the virtual avatar to the goal.

Team work is heavily encouraged due to the interdependence between the players: There is information asymmetry (section A.2.3) because the remote player can see some pre-existing blocks the local player cannot see, and the fact that local players can place new blocks in the game world provides ability asymmetry (section A.2.3).

A.3.3 FPS Survival game

The goal of the game is to protect a virtual structure from enemies. These enemies will come from multiple sides to attack the structure while the remote players have to stop them. For doing so, they require the aid of the local players, who can place blocks (similar to the platformer game, section A.3.2) to block the path of the enemies.

The local players can block the path of the enemies, but they cannot defeat them. The remote players, who view the game from a first-person perspective, have to defeat the enemies. Since the perspective of the remote players is limited (also partly because of the blocks placed by local players), they require information from the local players, who view the scene from above and as such have a good view of the entire situation. This creates information asymmetry between the players (section A.2.3).

To keep the game challenging, the enemies will grow stronger over time, requiring the cooperation between the local and remote players to improve as well. Because of the fact that local players can only defeat the enemies, and the remote players can block their path, there is a form of ability asymmetry between the local and remote players (section A.2.3).

A.3.4 Tower defense

The local players start off with a number of towers each which have certain strengths and abilities. They must place these towers on a table in front of them. What they cannot, but the remote player can see is what paths a number of hostile entities will take in order to attack their base. They must therefore cooperate to place the towers on the right positions to be able to defeat the enemies before they escape by communicating the ideal locations of the towers and paths that the hostile entities will take.

This game idea is relatively simple, and because of the simplicity, complexity can easily be added to make the game more engaging. For example, towers could be upgradeable, utilizing resources that could be gained either by defeating individual enemies or by defeating a wave of enemies. The idea behind these resources is that they are shared between players. They could then be used to upgrade tower types. As these resources are shared, and they can only be used once, players must work together to choose the best upgrade available in the given scenario. Enemies should also get procedurally stronger because of the increased capabilities of the towers. Resources could also be put towards research for new tower types, which players could then use.

Another way to add extra necessary complexity would be through introducing an experience system, and to grant towers some experience based on what enemy they have defeated. These towers would also over time get stronger. Bloons Tower Defense, a game found on the internet developed by Ninja Kiwi, is a tower defense game that uses both resources and experience, to show that these could also be combined. New towers would then be unlocked over time, after completing a certain amount of waves, instead of using resources.

The problem with this idea is that it will very likely create a *puppet master* scenario where the remote player takes charge of the whole situation and the other players do net get a say in what is going to happen. This could be avoided by a number of extra abilities. This game shows heavy information and ability asymmetry.

A.3.5 Minesweepers

This concept is based on the classic game of minesweeper. There's a grid where some of the squares have mines under them. Players each start at a random position and have to place flags on locations of mines while avoiding mines. The remote player is the only one who can see the numbers around squares, so he has to give instructions to the players on the field.

The difference compared to the classic version of the game is that it's all physical. Local players walk around on the field, where they have to take careful steps to avoid triggering a mine. This turns the game into some sort of Twister variant where people can use special moves to quickly traverse the field and find all the mines. Local players co-operate by dividing the field into sections that they'll clear in parallel.

The problem with this concept is the inherent *puppet master* phenomenon. Communication between the remote player and local players is very one-sided. Local players just receive commands where to step and where to plant flags and don't really have any input into the game themselves. Even if that problem was solved, the lack of cooperation aspects between local players would also be a big problem. Local players don't really have any reason to interact with each other. One way to solve that problem would be to require multiple players to place a flag.

There are also some technical challenges with this concept. It requires that local players always know their exact position on a large field that they're inside of, which would require a lot of markers. Next, the position of their legs would have to be determined somehow to ensure that they're not stepping on a mine.

Finally, a space large enough for a field would likely have to be found outside and sunlight doesn't play well with augmented reality devices. The game could be transformed into an indoor variant instead, using a board and pawns, but this does not solve the one-sided communication problem.

A.3.6 The chosen idea

The idea that is chosen is the mirror game idea. The reason for this decision is that it is a relatively simple idea, it can create hard to solve puzzles, and extra complexity can easily be added. It also prevents a *puppet master* scenario by giving both types of players about half of the abilities required for solving the game.

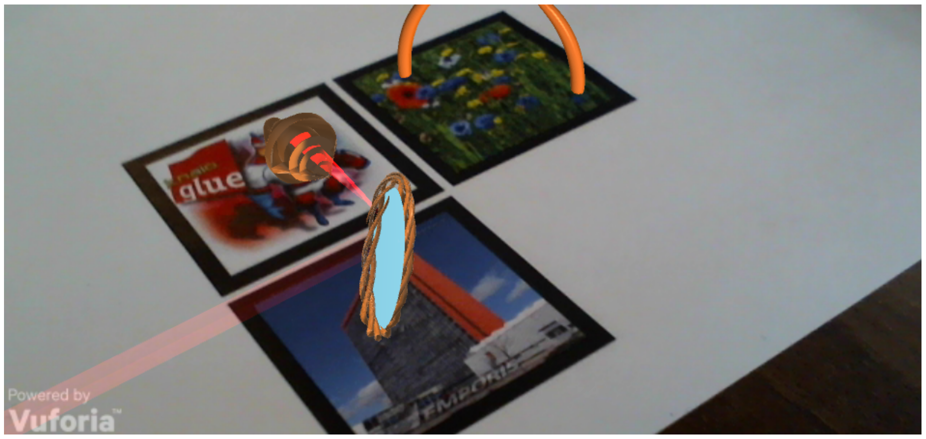
The game offers a strong interdependence between players. This, together with the ability asymmetry caused by the separation between moving mirrors and rotating them, causes a high need for collaboration between the local and remote players. As such, this concept comes closest to the problem formulation in creating a collaborative game between physically remote players.

A.4 Conclusion

In short, the mirror game seems the best solution to the problem. It is a viable project to create in the given time frame, it is a challenging puzzle game even when playing a different version of it (a single-player version) on your own, and it taxes the communication of both the remote and the local players, as they both have abilities necessary for achieving the goal, but they cannot achieve it on their own (although this could be relaxed to lower the entry barrier). Also, it is still a challenging project, because of the technical challenge of integrating AR hardware, recognition of markers with the game world and synchronization with the remote player. Additional complexity, such as colored laser beams/targets, beam splitters, beam mergers etc. could also be implemented to increase the technical challenge of the problem.

The game and networking will be implemented using Unity, with the Vuforia library handling augmented reality for the local players. The remote players will likely be using an Oculus Rift and the local players one of the augmented reality devices describes in the problem analysis that ends up working best in practice.

Using this information, we've built a small demo that uses Vuforia. It places a laser emitter, mirror and wall on three markers that can be moved around. The mirror reflects the laser realistically based on the angle of incidence. We've used this demo to test augmented reality setups with different types of markers, hardware and frameworks.


Powered by
Vuforia

Appendix B

Product plan

B.1 Concept

This section covers the concept idea of the gameplay, and the accompanying requirements.

B.1.1 Gameplay

The goal of the game is to solve a puzzle by controlling laser beams using mirrors in such a way that a predefined target is hit. The game can be played by one or more local players and one or more remote players.

There are cards present for the local players that represent mirror bases. These must be placed on the table, which will be the locations for the mirrors. The local players will be able to see the mirrors they place through the use of AR technology. Each of the local players will only be given a few of the mirror bases needed to solve the puzzle, and as such solving the puzzle requires cooperation from all local players.

The remote players can also see the placed mirrors, and can rotate them to influence the path of the laser beam(s). Only by cooperation between local players (who can only move the mirror bases) and remote players (who can only rotate them) it becomes possible to hit the target and as such solve the puzzle.

The game provides various different types of mirrors with different properties, allowing for more complex puzzles. One example of such a mirror is a colored mirror, and then require the target is hit with the right (combination of) colors. Another way to make puzzles more complex is requiring that the players combine beams together to create more powerful beams. Other optical components like beam splitters can also be introduced.

B.1.2 Requirements

Must haves

- Light source, a target and zero or more blocks must be visible when the game starts.
- Light source must emit a light beam in a predefined direction.
- Laser beam must reflect against a mirror when it hits one.
- Laser beam must stop when it hits a block
- Local player must be able to see mirrors positioned on the cards using AR technology.
- Local player must be able to move the mirrors by moving the cards.
- Remote player must be able to see the mirrors in the same positions and orientations as the local players.
- All players must be able to see the laser beam(s), the light source, the target and the blocks.
- Remote players must be able to rotate the mirrors.

Should haves

- Elevations of mirrors, to allow light of a lower elevation to hit the target on a higher elevation, or the other way around
- Light beams should be colored (after a certain level), and only light beams of a certain color can suffice in hitting the target.
- Combiners (think AND- or OR-gates) that combine light and cause it to travel to the target.
- Color combiners, to allow for a broader variety of color-specific targets.

Could haves

- Infinite amount of levels.
- Random level generation according to maximum difficulty settings.
- Give hints if the players are stuck.

Won't haves

- Playable with only remote players.
- Playable with only local players.
- Playable on Android or iOS devices.

B.2 Approach

This section covers development details that aren't directly related to the gameplay itself, such as software engineering practices and guidelines about client meetings.

B.2.1 Technical details

Although AR glasses have been provided to us, their field-of-view is very limited and is not suitable for most of our concepts. We experienced ourselves that the limited fov causes a lot of problems. Instead, we're going to try to use an Oculus Rift and cameras to create our own high fov augmented reality glasses.

The Unity game engine is going to be used to render the in-game objects like mirrors and laser beams over the camera image.

B.2.2 Software engineering

Using the Unity game engine means that we'll use a loosely coupled component-based architecture from the start. Unity has built-in unit and integration testing systems, which we'll make use of in development. We'll also maintain UML diagrams of the architecture to keep an overview of how everything works, and to plan integration of new features.

We'll make use of the Scrum software development methodology to plan new features and to help stick to the schedule. As is expected, there will be a playable demo at the end of each weekly sprint. As soon as the game has all the basic ingredients to be fun to play, we'll find users to play test on a regular basis and collect feedback from them to improve the gameplay.

The version control software used is Git. The project is on GitHub. Git uses a master branch (the basic branch which contains the project), as well as multiple other, user-defined branches. The branch structure of Git allows a team to work in separate branches, to merge the branches into the master branch later when that part of the project is done. This is done by making a pull request, defining what was changed, and afterwards the branch can be merged with the master branch. The way branches work also allow other team members to do code reviews of a branch, as all changed/added code can be seen and commented on.

B.2.3 Guidelines

Here is a list of rules that help prevent problems during development.

- Meet with the client and coach every week to show a working demo
- Add tests as soon as new methods are added to verify that they work
- Have integration tests for common scenarios to reduce the need for user testing
- For C# code, we adopt the guidelines presented by Microsoft (See `https://msdn.microsoft.com/en-us/library/ms229042.aspx`)
- Work on the project in the INSYGHTLab from 9 to 5 every workday (except for the dates listed in the planning section)

B.3 Planning

The first two weeks represent the research phase. In this phase we will find a suitable augmented reality (AR) library for Unity and prepare an Oculus Rift for AR use with cameras. We'll also design an architecture for the game that covers the marker detection, networking and gameplay mechanics. All of this information will be described in the research report handed in on May 1st. Main development commences after this phase and is organized in weekly sprints. The table below describes the goals per sprint, which will serve as a helpful reference to stay on schedule during development.

The first SIG submission is due May 26th. By then, we should already have a semi-functional product. The basic functionality (AR functionality, basic gameplay mechanics, etc.) should be in this version of our product. All the relevant graphics assets should also be done.

The second SIG submission is due June 6th. By then, most of the functionality should be implemented (this includes advanced gameplay mechanics, hitting multiple targets, mirror elevations, etc). Also, all the bad parts of the code structure/architecture from the first version should be resolved.

The project should be done by June 26th, the Friday of week 4.10. This includes not only the project but also a final report of 40 to 50 pages. This means that the report should be worked on continuously.

Weeks	Deadline	Goal
4.1 + 4.2	May 1st	Research report described above
4.3	May 8th	AR integration + lasers
4.4	May 15th	Basic gameplay start + start report
4.5	May 22nd	Finalize first SIG version
4.6	May 29th	Start advanced gameplay
4.7 + 4.8	June 12th	???
4.9 + 4.10	June 26th	???

B.3.1 Holidays + businessdays on which EWI is closed

Koningsdag	27-04-2015
Dodenherdenking	04-05-2015
Bevrijdingsdag	05-05-2015
Hemelvaartsdag	14-05-2015
De dag na Hemelvaartsdag	15-05-2015
2e Pinksterdag	25-05-2015

Appendix C

SIG Midterm Evaluation

De code van het systeem scoort ruim vier sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size, Unit Complexity en Unit Interfacing.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'detector::recognizeMarkers'-methode, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld '// Turn into grayscale and threshold to find black and white code' en '// Cut off border' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methoden ook naar voren als de langste methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Wat opvalt in dit systeem is dat zowel in de C# als in de C++ code soms een Point/Vector abstractie gebruikt wordt, maar dat er ook methoden zijn waar de parameters 'x' en 'y' los worden doorgegeven. Om het voor toekomstige ontwikkelaars makkelijker te maken om de code te hergebruiken is het aan te raden de abstracties consistent te gebruiken.

Daarnaast nog de opmerking dat het goed is om te zien dat de README duidelijk aan geeft dat de 'netlink' code niet zelf geschreven is. Zou het hier nog helpen om duidelijk aan te geven welk versienummer van deze library nu in gebruik is?

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

Appendix D

SIG Final Evaluation

In de tweede upload zien we dat het codevolume met ongeveer 50 procent is gestegen, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven.

Bij Unit Size en Unit Complexity valt op dat de deelscores voor de C++ code zijn gestegen, maar voor de C# code zijn gedaald. Die daling wordt vernamelijk veroorzaakt door een aantal lange en complexe methodes die jullie sinds de vorige upload hebben toegevoegd. Voorbeelden zijn MarkerTransformer.Update en ClientSocket.ReadMessage. Aan de namen van die methodes kun je al zien dat ze erg veel doen, en het opsplitsen in deelproblemen zou deze methodes beter onderhoudbaar en vooral beter testbaar maken.

Bij Unit Interfacing zien we geen structurele verbetering voor zowel de C# als de C++ code. Zoals bij de eerste upload al aangegeven wijst dit vaak op een gebrek aan abstractie. In MarkerTracker komen bijvoorbeeld vier methodes voor die allemaal dezelfde lijst van vier parameters aan elkaar doorgeven.

Tot slot is het goed om te zien dat jullie sinds de vorige upload unit testcode hebben toegevoegd. De hoeveelheid testcode is nog wel vrij beperkt, maar dit is wel te verklaren aangezien jullie vrij laat begonnen zijn.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie deels zijn meegenomen in het ontwikkeltraject.

Appendix E

Info sheet

General information

Project title: Collaborative AR mirror game
Name of the client organization: TU Delft
Date of the final presentation: July 3, 2015
Final report: TBA.

Description of the project

Although augmented reality research has grown into a mature field over the last years, the aspects of situational awareness and presence of augmented reality are still open research topics. The mirrors AR game is a collaborative game that can be used to explore different perceptions of situational awareness and the effect on solving puzzles.

The main challenge of this project was to develop a system that intuitively lets co-located and remote players work together on a problem. Most of the research phase was dedicated to finding the elements that make this feasible. For example, we gave each player limited abilities so that they're forced to work together to reach a solution. That way no player will feel like they're expendable. The biggest issue we ran into is that the AR glasses we're using have abysmal tracking performance, which made it more difficult for local players to be immersed.
In the game players have to guide one or more laser beams from emitters to targets by placing mirrors inside a level. Each level poses new challenges by including walls and optical components like beam splitters.

There are two kinds of players: local and remote. Local players wear see-through AR glasses and see the level projected onto a surface in a room. They can walk around and place so-called markers on the table to indicate where in the level they want to place a mirror. Remote players are not physically co-located and see just the level and mirrors on their computer screen. However, only they have the power to rotate mirrors and a good overview of the entire level. The players will need to cooperate to get all of the mirrors in the right places with the correct orientation to complete the level.

The mechanics and restrictions of the game can be controlled to research the effects on collaboration. Examples:

- Viewpoint of local players can be shown or hidden to remote players.
- Remote camera could be locked or be allowed to move freely.
- Hide the level for either the remote or local players to demand more communication.

The game can support any number of players as long as there's at least one local and one remote player.

Members of the project team

Name: Thijs Boumans (T.Boumans-1@student.tudelft.nl)
Interests: Computer Graphics, Algorithm Design
Contribution and role: Front-end Developer

Name: Patrick Kramer (ptrck.krmr@gmail.com)
Interests: Software Engineering, Quality Assurance
Contribution and role: Lead Software Designer, AR Projection Mechanics

Name: Alexander Overvoorde (overv161@gmail.com)
Interests: Computer Graphics, Software Engineering
Contribution and role: Remote Player Visualization, Game Mechanics, Server Computer Vision

Name: Tim van Rossum (trvanrossum@gmail.com)
Interests: Algorithm Design, Software Engineering
Contribution and role: Final Report curator, Level Designer, Co-lead Software Designer

Coach and client

Coach: Rafaël Bidarra, Computer Graphics, r.bidarra@tudelft.nl
Client: Stephan Lukosch, Multi-Actor Systems, s.g.lukosch@tudelft.nl

Bibliography

[Endsley et al., 2003] Endsley, M., Bolte, B., and Jones, D. (2003). *Designing for situational awareness: An approach to user-centered design.* Taylor and Francis.

[Google, 2015] Google (2015). Cardboard. `https://www.google.com/get/cardboard/`, Accessed 01-May-2015.

[Lindeijer, 2014] Lindeijer, T. (2014). Tiled map editor. `http://www.mapeditor.org/`, Accessed 23-June-2015.

[Lukosh, 2015] Lukosh, S. (2015). Collaborative augmented reality tower game. `http://bepsys.herokuapp.com/projects/view/82`, Accessed 23-April-2015.

[MacIntyre, 2012] MacIntyre, B. (2012). Unity ar toolkit. `https://research.cc.gatech.edu/uart/content/about`, Accessed 28-April-2015.

[Meta, 2015] Meta (2015). Meta one. `https://www.getameta.com/`, Accessed 28-April-2015.

[Metaio, 2015] Metaio (2015). Metaio. `http://dev.metaio.com/sdk/tutorials/see-through-glasses/`, Accessed 28-April-2015.

[OculusVR, 2015] OculusVR (2015). Oculus rift development. `https://developer.oculus.com/`, Accessed 28-April-2015.

[Qualcomm, 2015] Qualcomm (2015). Vuforia. `https://www.qualcomm.com/products/vuforia`, Accessed 28-April-2015.

[Sutherland and Schwaber, 2013] Sutherland, J. and Schwaber, K. (2013). *The Scrum Guide: the definitive guide to Scrum, the rules of the game.*

[Visualead, 2015] Visualead (2015). Graphical qr codes. `http://nl.visualead.com/`, Accessed 01-May-2015.

[Zagal et al., 2006] Zagal, J., Rick, J., and Hsi, I. (2006). Collaborative games: lessons learned from board games. *Simulation & Gaming Vol.37 No.1.*