Portfolio

accompanying the report “Increasing Presence in a Virtual Environment by Adding Physical Objects

MTA17531

Aalborg University - Medialogy

Contents

Preface

Ronny Vangsted
<rvangs15@student.aau.dk>

Andreas Hejndorf
<ahejnd15@student.aau.dk>

Rasmus Bugge Skammelsen
<rskamm15@student.aau.dk>

Robert Wittmann
<rwittm15@student.aau.dk>

Balázs Gyula Koltai
<bkolta15@student.aau.dk>

Mózes Adorján Mikó
<mmiko15@student.aau.dk>

1. Concept

It was decided from the beginning of the project, that the focus is virtual reality. The product of this project is a game set in a virtual reality environment. After being presented several project proposals, it was decided that the project will focus on testing what effect physical objects have on presence in a virtual reality environment expanded beyond the borders of the physical play area.

During background research, it was found that there is a lack of research on the use of trackers to implement physical objects in virtual reality. Therefore, this became the main focus of this semester project. Through research, a technique called "self-overlapping architecture" was found. With this, a project consisting of multiple different spaces could be created while using the same physical area. The research can be found in chapter 2 of the report.

1.1 Process

As the project aimed to create interactive physical objects in an environment made with self-overlapping architecture, it was necessary to design a scene where interacting with the environment in different ways was a natural part of the experience. This scene will be used as a tool to perform an experiment, that demonstrate how well physical interaction can be applied to virtual reality environments.

With the project consisting of object interaction and multiple rooms, themes that would naturally lend themselves to such a setting were considered during brainstorming; the group discussed different ways of movement, different ways of expanding the perceived play area, different concepts, and different tests.

The most interesting ideas were narrowed down to two: a dungeon crawler and an escape room-style game. In order to utilize change blindness the user needed to be distracted, so they would look away while the door changed position: in a dungeon crawler enemies could appear in the general direction that would coerce the user to look away from the door, and in an escape room a clue or a tool could be put in the opposite corner. The group decided to go with the escape room as the forms of interaction in such a setting would be near limitless, whereas for a dungeon crawler the genre limits the actions to a set amount. In an escape room the users' focus would also be channeled more towards the interactive objects in the environment.

The goal for the user is to navigate through the virtual rooms by solving puzzles in one area before being able to move on to the next one. Inspiration for the general

concept include the sequential puzzle narrative from The Da Vinci Code novel and puzzle games such as the Professor Layton series for Nintendo hardware [1].

As the concept was decided to be an escape room, it was necessary to decide on the theme and visual aesthetics of the setting. During brainstorm several different settings were brought up:

- Science fiction
- Prison
- Dungeon
- Vaporwave
- Steampunk
- Holidays
- Aquatic landscape
- Office building

Several of these ideas were scrapped because they did not fit the group's idea for the project. The aquatic landscape, vaporwave, and science fiction were decided against because the aesthetics would have complicated the design of the environment, the objects or the possible narrative for the project. In the end two settings were considered: the prison and the dungeon. The prison would have an obvious narrative, but the setting limited the puzzles, as not everything would make sense to be in a prison. A lot of the possible prison puzzles would require small tools, which would not be suitable for the interactions envisioned for this project. With inspiration from Indiana Jones, a dungeon could have many clever and elaborate puzzles. It was decided that the dungeon would be set in present time, so that more modern objects and tools could be used for solving the puzzles. This would provide some familiarity to the puzzles, so the user would not need to learn how the tools worked before learning how to solve the puzzles themselves.

This means that going with the chosen aesthetics and having the game take place in present day, we can have better control of the lighting with the inclusion of objects such as a construction light and a flashlight, the latter which is a key component of some of the puzzles in the project.

1.2 Room concept

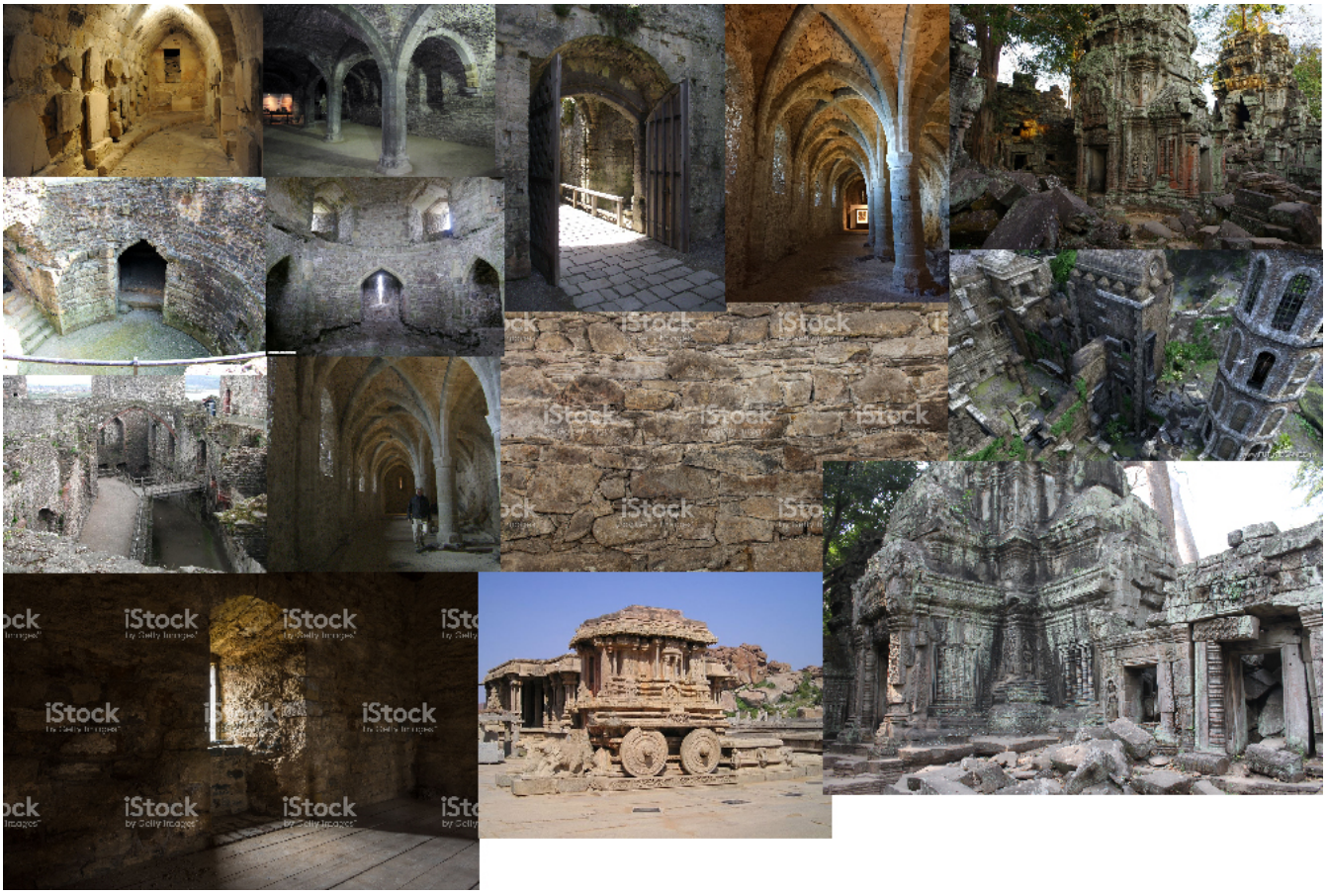


Figure 1.1: The mood board used in this project.

The moodboard in Figure 1.1 was developed to keep a consistency in the design of the different models as well as working as a sort of inspiration for the puzzles.

1.3 Puzzle concept

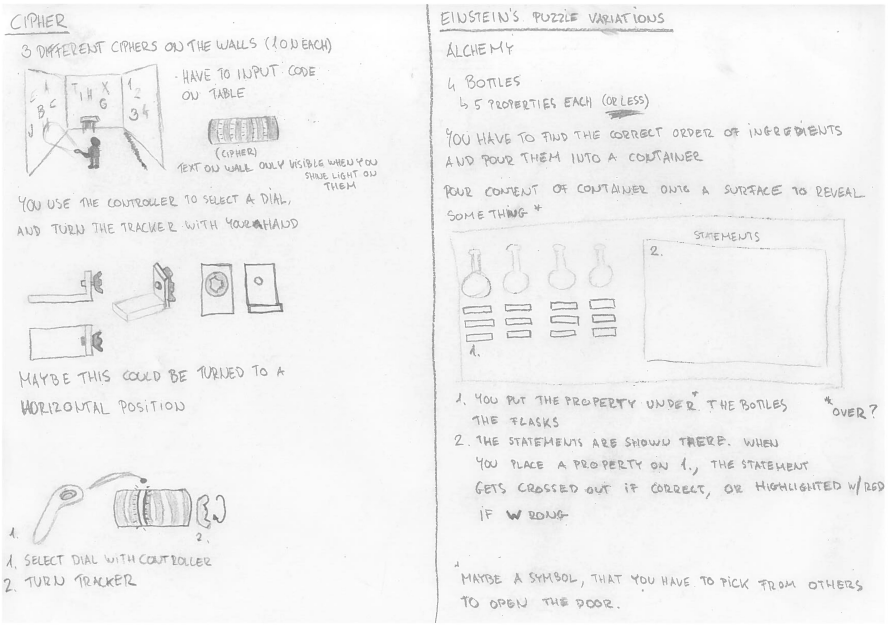


Figure 1.2: Concepts of cipher decoder and Einstein puzzle.

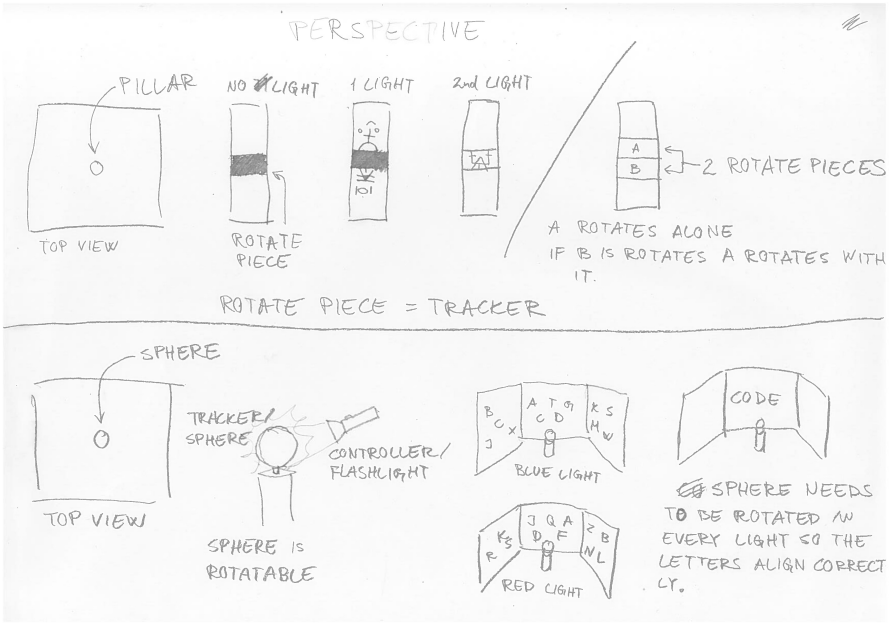


Figure 1.3: Concepts of the rotating pillar and light sphere.

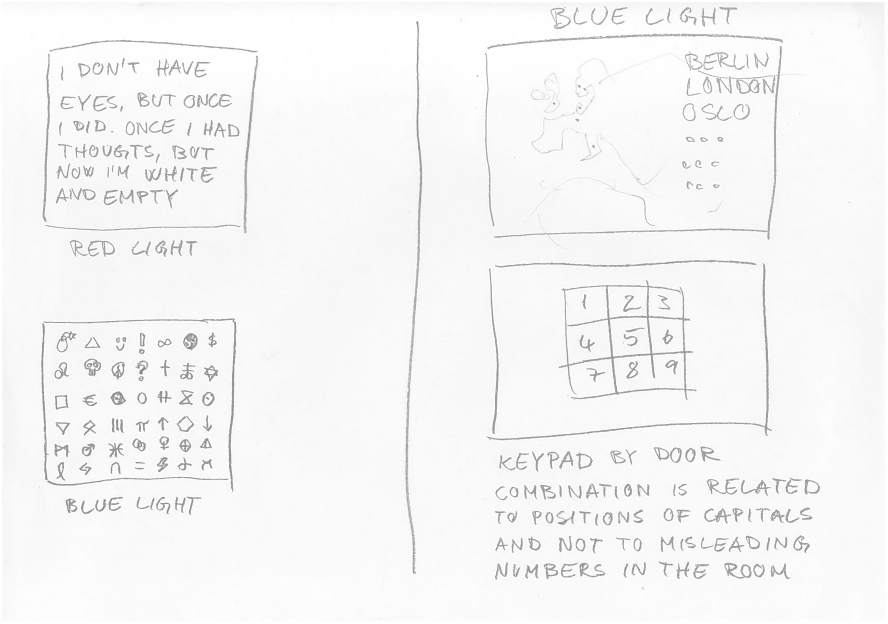


Figure 1.4: Concepts of the riddle and geography key-pad.

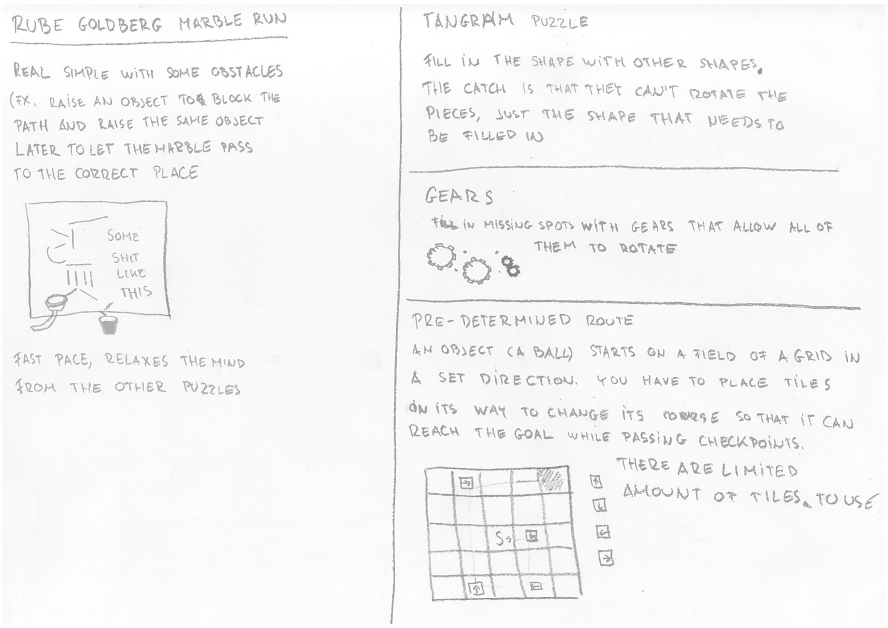


Figure 1.5: Concepts of the Rube Goldberg and tangram puzzle.

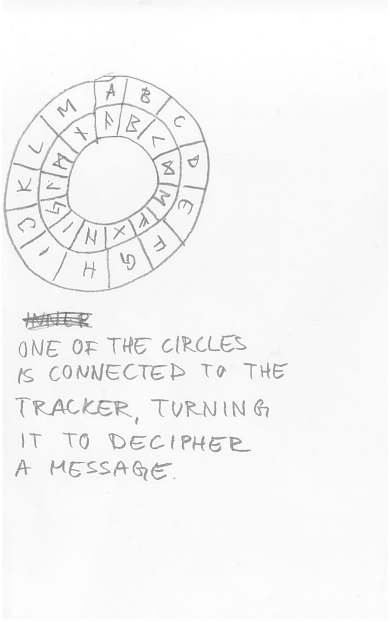


Figure 1.6: Concept of the cipher wheel.

2. Architecture

The virtual environment available for this project had a maximum size of 4 meters by 4 meters. For this project, one milestone was to expand the virtual environment beyond the physical play area within this 4 meters by 4 meters space without making any interruptions in the experience, such as standing in an elevator or teleporting. For this, research was conducted on other papers that had used the concept of change blindness to accomplish the same thing. This chapter goes into detail with how the final version of the design for this project was reached, and what thoughts went into it.

2.1 Design

The idea of overlapping architecture was first encountered in "Self-Overlapping Maze and Map Design for Asymmetric Collaboration in Room-Scale Virtual Reality for Public Spaces" by Evers, Serubugo, Skantarova [2]. Sömmad, the game that resulted in the research in that paper, uses self-overlapping architecture to create a maze. After researching and trying this design, it was concluded that the concept was promising, but it felt too much like going in circles. It became too obvious that the user kept walking in the same space, which would decrease presence.

"Leveraging change blindness for redirection in virtual environments" by Suma et al [3] also experimented with self-overlapping architecture, but have also tried to take advantage of change blindness. Change blindness is a concept where participants will not notice changes in the environment if they are distracted by something else. The self-overlapping design from that paper, altered to fit the space available for this project, can be seen on Figure 2.1. On all the figures in this chapter, each dark square represents 1 meter by 1 meter. Each room is 3 meters by 4 meters and the corridor is 1 meter by 4 meters. Whenever the user is distracted by a strategically placed object in the corner, the door goes to the opposite side of the corner and the corridor follows. When utilizing change blindness in this way, it appears to the users as if they are traveling down a straight corridor with doors to their left. Technically this process could be repeated infinitely - the paper above made a 60 feet corridor with 12 rooms.

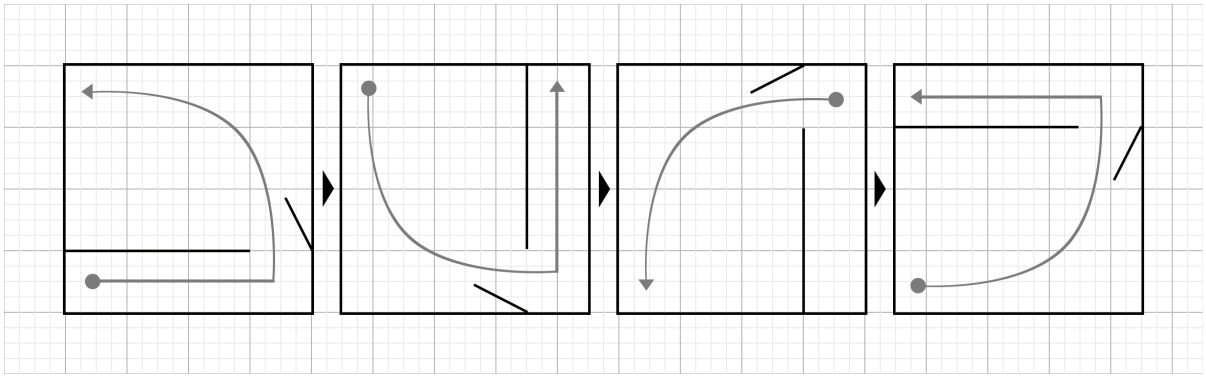


Figure 2.1: The design of overlapping architecture and impossible spaces used in [3], but altered to fit the space in this project.

This design fit nicely with the goal of this project, as it eliminated the problem found in Sömmad. It also maximizes the size of the room, as the corridor only takes up space in one side of the room at any time. Incidentally, the problem arose because of this shifting corridor when it was applied to this project. As it can be seen on Figure 2.1 the size and shape of the rooms changes dramatically, when the corridor changes place. This does not affect change blindness, but it does limit the scope of what can be done, when the room needs to be inhabited by physical objects. As the walls change place in two sides of the room any objects would have to be placed away from them: any object next to the wall that gets pushed into the room would be clipped inside the wall. The wall that gets pushed out would create a space between the wall and the object, and as no research has been done on how this affects the illusion of infinite space, it was concluded it was best to not do this. This would not be a problem had the objects just been virtual, as they could just be moved with the walls, but because this project integrates physical objects with trackers, it was deemed that this design was not feasible.

Even though the above idea was discarded, inspiration was drawn from it, as the never-ending corridor was very well suited for this project, it just had to be adjusted so that physical objects could be placed within it. This meant that the room had to have the same size and dimensions at all times. The result of this was the room design seen on Figure 2.2. This design uses the same concept as the one on Figure 2.1, but it allocates space for corridors on all sides of the room at all times. This way the room stays the same, no matter what corridor the user is in. It does however decrease the size of the room. In this design the corridors are 0.5 meters wide, meaning that the room is 3 meters by 3 meters instead of 4 meters by 4 meters. This design still was not perfect though.

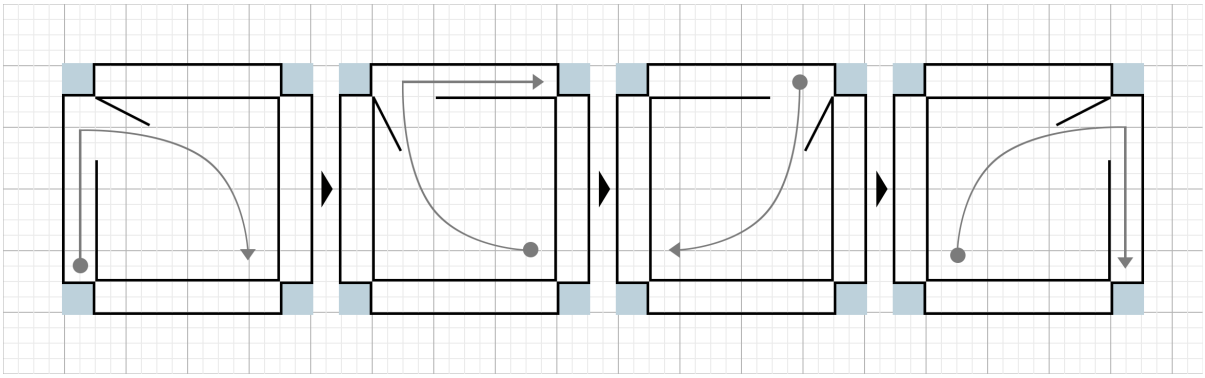


Figure 2.2: The overlapping architecture layout designed for this project.

While it utilized the physical space well, it was discovered while play-testing the design, that the wire connecting the head-mounted display to the PC was too short and became taut when the user tried to reach the far corner of the room. It was also determined that the corridor was too narrow to comfortably navigate through. With these shortcomings identified the room on Figure 2.3 was designed.

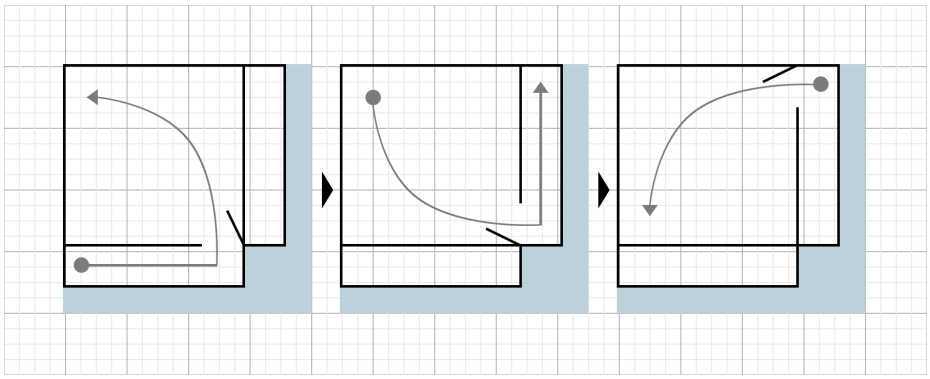


Figure 2.3: The variation of the overlapping architecture from Figure 2.2 that was used in this project.

For this project it was only necessary to create two rooms within the physical play area. Because of this, it was possible to remove the two unused corridors, and push the room itself further into the corner. Doing this in the environment in Figure 2.2 would make the diagonal length from the corner of the play area to the corner of the room 0.7 meters shorter. To make the the corridors a more maneuverable size the width was changed from 0.5 meter to 0.625 meters. This was done because this was the width used in "Self-Overlapping Maze and Map Design for Asymmetric Collaboration in Room-Scale Virtual Reality for Public Spaces" and this was found acceptable. The architecture design on Figure 2.3 is the one used in the artifact for this project.

2.2 Implementation

The overlapping architecture is handled in three classes. The first is the GridGenerator which can be seen in code snippet 2.1. This class makes half meter by half meter tile prefabs, arranges them around the center in a specified size, in this case four meters by four meters and assigns a unique ID to each tile.

Code 2.1: Grid generation method in the GenerateGrid class

```
1  public void GenerateGrid()
2      {
3          for (int x = 0; x < gridSize.x; x++)
4          {
5              for (int y = 0; y < gridSize.y; y++)
6              {
7                  Vector3 tilePosition = new Vector3(transform.position.x +
                       -gridSize.x * (tileScale / 2f) + tileScale / 2f + x *
                       tileScale, transform.position.y, transform.position.z -
                       gridSize.y * (tileScale / 2f) + tileScale / 2f + y *
                       tileScale);
8                  Transform newTile = Instantiate(tilePrefab, tilePosition,
                       Quaternion.Euler(Vector3.right * 90)) as Transform;
9                  newTile.transform.localScale = Vector3.one * tileScale;
10                 newTile.GetComponent<Tile>().SetID((int)gridSize.y * x + y);
11                 newTile.parent = transform;
12             }
13         }
14     }
```

This grid is then used in the PlayerTracker class, which can be seen on code snippet 2.2. The PlayerTracker takes the transform the head-mounted display, and casts two rays from its position. One straight down and one where the HMD is facing. The GetTileID() method receives the ID of the tile below the user, while the LookingAtTile() gets the ID from the tiles not generated by the grid, but placed vertically by us in the rooms. The difference between the two methods is the Layermask used, so one of them can only register "floor tiles", and the other can only register "direction tiles". Both the location and direction values are stored in a LocDirID struct.

Code 2.2: Method for getting the tile ID in the PlayerTracker class

```
1  void GetTileID()
2      {
3          Ray ray = new Ray(trackedObject.transform.position, Vector3.down);
4          RaycastHit hitInfo;
5          int tileMask = LayerMask.GetMask("Tile");
```

```
6
7           if (Physics.Raycast(ray, out hitInfo, 100, tileMask))
8           {
9               Debug.DrawLine(ray.origin, hitInfo.point, Color.red);
10              locationDirection.tile =
                    hitInfo.collider.GetComponent<Tile>().GetID();
11          }
12          else
13          {
14              Debug.DrawLine(ray.origin, ray.origin + ray.direction * 100,
                    Color.green);
15           locationDirection.tile = -1;
16          }
17      }
```

In the RoomSwitcher class, seen on code snippet 2.3 and 2.4, a reference to the PlayerTracker is made, and the current location and look direction IDs are constantly updated to the RoomSwitcher's "locationDirection" variable. An array of GameObjects is also made, where the rooms and corridors of the game can be put. When the game first starts, there is a special condition, which is to stand on a calibration zone that will enable the first two areas, namely Corridor1 and Room1. After this, whenever a change condition is met, the area before the current one is disabled and the next one is enabled. Since there are two rooms, this happens twice in the game, but can be extended indefinitely.

Code 2.3: The update method from the RoomSwitcher class, with the starting and regular switch conditions

```
1   void Update()
2       {
3           locationDirection = tracker.GetLocationAndDirection();
4
5           if (currentRoom == 0)
6           {
7               if (CheckCondition(startPosition))
8               {
9                   StartRooms();
10                  currentRoom = 2;
11              }
12          }
13          if (currentRoom != 0 && currentRoom < maxRooms - 1)
14          {
15              if (CheckCondition(switchCondition[currentRoom - 2]))
16              {
17                  SwitchRoom();
18                  currentRoom++;
```

```
            }
        }
    }
```

Code 2.4: StartRooms() and SwitchRoom() methods from the RoomSwitcher class

```
void StartRooms(){
        Room[0].SetActive(false);
        Room[1].SetActive(true);
        Room[2].SetActive(true);
    }

    void SwitchRoom()
    {
        if (currentRoom < maxRooms - 1 && currentRoom != 0)
        {
            Room[currentRoom - 1].SetActive(false);
            Room[currentRoom + 1].SetActive(true);
        }
    }
```

3. Experiment 1: Hand Awareness

One of the key elements of the project is the interaction with virtual objects compared to interaction with physical objects in the virtual environment. Before designing the multiple puzzles that provided interaction for the participants, it was necessary to know how many trackers could be allocated for this purpose.

A small experiment was designed and conducted to test which method for interaction was preferred for physical and virtual objects respectively. Aalborg University has four trackers, all of which were provided for the project. The main concern was whether it is important for the users to be able to see their hands in VR to be able to interact with objects, as that would mean that two trackers could not be used for the puzzles. 16 participants performed a small test, testing both accuracy of interaction under different conditions, as well as the participants' preferred method of interaction.

3.1 Hypothesis

The hypothesis that is being tested is the following:

> *Users with their hands tracked in virtual reality perform better in motor precision tests.*

This means that the null-hypothesis that is being tested is:

> *Users with their hands tracked in virtual reality do not perform better in motor precision tests.*

3.2 Procedure

The following procedure was used during the testing:

1. Pick up an object
2. Walk around the table
3. Place object within a circle on the table

The translation and orientation of the tracker compared to the circle will be measured. The VR setup is showed in Figure 3.1.

Figure 3.1: The VR setup of the test. The task is to move the tracker (left) into the circle (right) with respect to the colours.

Each participant will go through the following two scenes:

1. Version - Virtual: The objects are virtual and the user has the VR controllers in their hands. The controllers can be made visible (V) and invisible (IV) at will.

2. Version - Physical: The objects are physical and tracked in VR and the user has VR trackers on their hands. The trackers can be made visible (V) and invisible (IV) at will.

To make sure that the learning effect will not affect the results, the participants will be presented with the four different scenarios as specified in Table 3.1.

Participant	**Task rotation**
1, 9	1V - 1IV - 2V - 2IV
2, 10	1IV - 1V - 2V - 2IV
3, 11	1V - 1IV - 2IV - 2V
4, 12	1IV - 1V - 2IV - 2V
5, 13	2V - 2IV - 1V - 1IV
6, 14	2IV - 2V - 1V - 1IV
7, 15	2V - 2IV - 1IV - 1V
8, 16	2IV - 2V - 1IV - 1V

Table 3.1: Table of task rotation

Before each test, the participant will be presented with their task. See appendix A in section A.1 to read the full script.

3.3 Pilot test

During the pilot test, which was conducted on one of the group members, the following observations were made, and taken into account for the actual test:

- Using an accurate representation of the trackers on the wrists in VR is better than using a model of a hand, as the movements of the hand can not be represented accurately enough.
- It was found that an exit-survey would be beneficial, because what the user prefers in terms of interaction may be more important than accuracy.
- Drifting from the trackers may cause the physical object tracking to be less accurate.
- Some changes to the script had to be made:
  - The participants have to walk around the table to the left, so the cable does not get tangled and move the table.
  - The participants have to stand in the middle of the VR-space before starting each part of the test.

After this, the test setup was changed so the participants would only see the trackers on their wrists, instead of a 3D model of a hand.

A post-test survey was created, where the participants would be asked for their opinion on the different interaction types, and what kind of interaction they preferred. For the full post-test survey, refer to appendix A in section A.2

3.4 Results

All numerical results from the test have been arranged in a table, and can be seen on Table 3.2. The answers from the post-test survey can be found in appendix A.

3.4.1 Data preparation

In order to prepare the data to be analysed, the translation and rotation offsets measured in the test need to be converted into one number that represents the total displacement. This number will be the translation offset + the angle offset converted into a distance, see equation (3.1).

$$d_{total} = d_{translation} + \frac{\alpha_{rotation}}{360} \times (2 \times r_{vive} \times \pi) \tag{3.1}$$

The Vive tracker has a radius of 50mm.

Participant		Virtual object		Physical object	
#		Visible	Invisible	Visible	Invisible
1	Translation Rotation	6mm 12°	7mm 0.7°	14mm 12°	17mm 2.1°
2	Translation Rotation	8mm 15.5°	6mm 0.4°	13mm 3.7°	19mm 1.6°
3	Translation Rotation	4mm 0°	10mm 0.8°	12mm 6.8°	12mm 6.6°
4	Translation Rotation	8mm 4.72°	6mm 5.9°	23mm 5°	17mm 1.4°
5	Translation Rotation	8mm 2.6°	7mm 11°	5mm 1.3°	13mm 2.9°
6	Translation Rotation	13mm 2.4°	4mm 2.7°	2.8mm 1.7°	13mm 3.2°
7	Translation Rotation	4mm 0.2°	8mm 4.6°	41mm 1.7°	13mm 2.2°
8	Translation Rotation	13mm 4.6°	2mm 2.5°	11mm 2.3°	12mm 2.2°
9	Translation Rotation	6mm 0.3°	4mm 4.4°	13mm 3.3°	12mm 2.2°
10	Translation Rotation	5mm 6.2°	8mm 0.2°	18mm 8.9°	14mm 8.4°
11	Translation Rotation	6mm 0°	0.8mm 1.8°	14mm 1.2°	13mm 3.5°
12	Translation Rotation	13mm 0.9°	7mm 28°	8mm 4°	11mm 2.2°
13	Translation Rotation	6mm 6.5°	3mm 1.7°	12mm 5°	15mm 3.5°
14	Translation Rotation	5mm 6.4°	4mm 0.2°	15mm 15°	12mm 1.4°
15	Translation Rotation	7mm 1.3°	6mm 3.3°	13mm 2.4°	13mm 1.3°
16	Translation Rotation	9mm 12°	13mm 8.3°	10mm 1.4°	16mm 3.8°

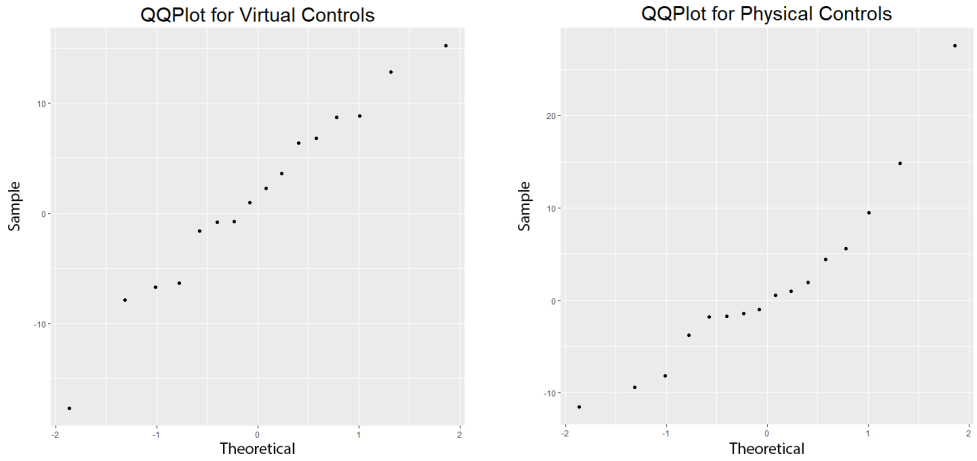
Table 3.2: Results from the hand awareness test

3.4.2 Assessing normality

Since two means are being compared, normality should be assessed in the difference between the two datasets. This means that the data we test for normality is the data for visible hands subtracted by the data for invisible hands - this is then done for both virtual and physical datasets.

To test for normality, the Shapiro-Wilks test was used with a significance value of 5%. The virtual data returned $W = 0.97, p = 0.90$, while the physical data returned $W = 0.90, p = 0.09$.

This means that the test resulted in both datasets being non-significantly different from the normal distribution, and hence both datasets can be considered normally distributed. However, as seen on the larger values from the virtual dataset, this has a higher p-value of 0.9037, meaning that it differs less from the normal distribution. To visually assess normality, qqplots were also created, see Figure 3.2



(a) QQPlot for virtual controls. **(b)** QQPlot for physical controls.

Figure 3.2: QQPlot of the virtual and physical datasets.

On both plots it can be seen that neither of the datasets represent an exact normal distribution, and that the virtual controls are closer to the normal distribution than the physical controls, which is also represented by the p-values, where virtual controls scored 0.9037, while the physical controls only scored 0.0861. Both datasets were deemed to be close enough to the normal distribution to be treated as parametric.

3.4.3 Testing the data

Since the data is parametric and there are two sets of data, where two means will be compared, a t-test will be conducted on both datasets.

On average, there were no significant difference in the precision of control between seeing the controllers (M = 11.70, SE = 2.93) and not seeing the controllers (M = 10.20, SE = 2.55), and it only represented a small effect size, r = 0.18. t(15) = 0.71, p > 0.05.

The results are similar for the physical data. There were no significant difference in the precision of control between seeing the trackers on the wrists (M =

18.24, SE = 4.56) and not seeing the trackers (M = 16.57, SE = 4.14), and it also only represented a small effect size, r = 0.18. $t(15) = 0.69$, $p > 0.05$.

3.4.4 Survey results

While the precision was not affected by the type of controller and visibility, the post-test survey revealed that the participants had some preferences. The participants generally agreed that they had no issues using a controller to interact with objects in VR, one participant pointed out that "*I know I'm in a game, so I'm perfectly fine with using a controller compared to my hands,*" while another participant stated that it was "*more immersive having a physical object in your hand.*" This is the problem that will be investigated further in the final product.

There were consensus among the participants that they preferred seeing the controller or tracker. One participant said it was "*very difficult as I had to guess/feel where my arm/hands were in relation to my body,*" when asked how well they were able to use their hands to interact with objects in VR.

Several of the participants also mentioned issues with the controller and trackers "drifting", which ruined the sensation of where their hands were.

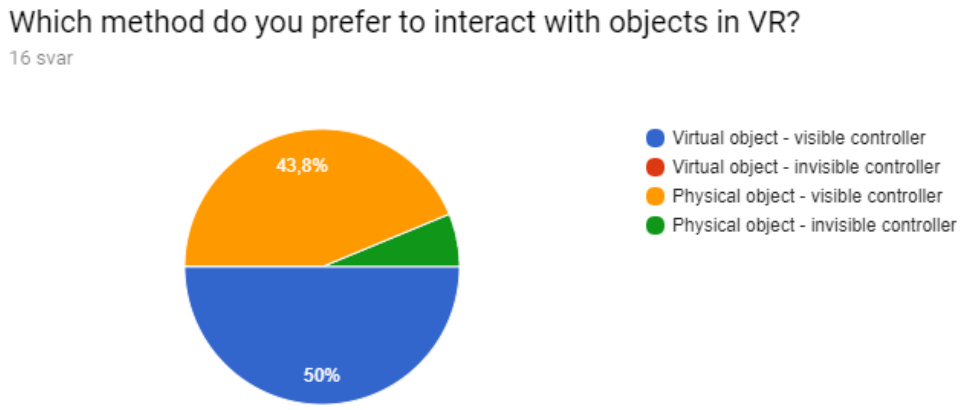


Figure 3.3: Piechart showing the participants preferences in control type.

In Figure 3.3 it can be seen that most of our participants (93,8%) preferred having visible controllers/trackers, while 50% preferred interacting with virtual objects and 50% preferred interacting with physical objects.

3.4.5 Conclusion

The purpose of this experiment was to find whether it is beneficial to be able to see your hands in a VR environment. It was concluded that there were no significant

benefit in the precision of interaction, when the participants could see the controllers/trackers in VR. However, we can conclude that 93,8% of the participants preferred to see the controllers/trackers, and therefore this will be implemented in the final solution.

It can also be concluded that the method used to attach the trackers to the wrists has to be improved, since the participants experienced severe drifting, and an unstable attachment. This could be because the light sensors on the equipment were blocked by the hands of the participants or the straps.

Finally, it was found that the participants were split in their opinions about interacting with physical objects or not, and this is what will be investigated in the final experiment

4. Puzzle design

Research for the project included researching various puzzles that could be used in the project. Inspiration was taken from existing escape rooms and puzzle games, but sketches for altered versions of popular puzzles were also made, as seen in section 1.3. The list of possible puzzles were the following:

- Cryptex
- Einstein's puzzle
- Rotate pillar
- Light sphere
- Riddle
- Map-keycode
- Rube Goldberg marble run
- Tangram puzzle
- Gears
- Pre-determined route
- Cipher wheel

From this list, 3 puzzles have been chosen to be included in the project: the Riddle, the Cipher wheel and the Light sphere. The rest were decided not to be included due to various limitations: the number of available trackers, need for additional interface or spatial limitation. Since the focus is on interaction with the environment rather than the puzzles themselves, it was not necessary for the puzzles to be very difficult. The main goal of the puzzles were to make them interesting and varied to interact with to promote presence in the virtual environment rather than the outside world. The process, final design, and implementation of the three puzzles and any accompanying objects will be described in this chapter.

4.1 Flashlight

4.1.1 Design

While not a puzzle, the flashlight is a crucial item for the project. It is used to reveal the riddle and different parts of the light sphere puzzle, similar to how a black light works. Contrary to traditional black light, the user can switch between three different colours using the touchpad on the Vive controller, and thereby reveal three different textures.

4.1.1.1 Virtual design

A Vive controller was used as reference to make sure the buttons on the controller and the model align correctly. The flashlight was modeled out of a cylinder with multiple subdivisions. The edge loops have been scaled and extruded where necessary to achieve the final shape. The buttons were modeled from a torus that was separated into 4 different but equally sized objects.

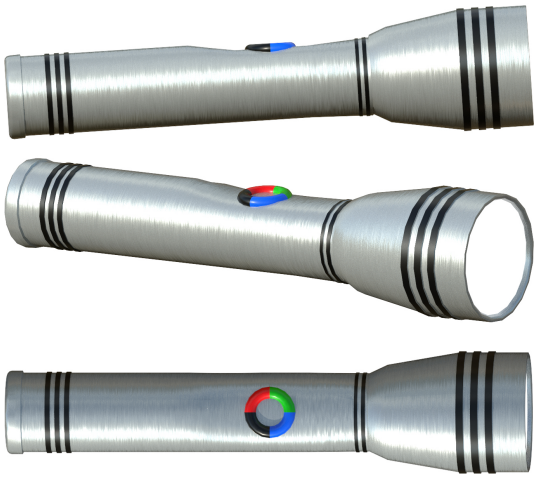


Figure 4.1: The model of the flashlight as seen in VR

4.1.1.2 Physical design

As the flashlight needed to send user input to the system, a tracker on a flashlight-shaped stick would not work. The object trackers do not receive direct user inputs, and making a custom button control would warrant a whole new set of tests of the design and implementation. As the Vive controller functions the same as object trackers with the added benefit of buttons and having the basic shape of a flashlight, it was decided to use that as the physical counterpart for the flashlight.

4.1.2 Implementation

The implementation of the flashlight consists of a shader and a Unity script.

4.1.2.1 Shader

We started out with a standard Unity surface shader. Because light is coming from a flashlight, the light source will be a spotlight. The image or writing that we want to reveal is added to the surface as a decal.

We need to know the position, direction and angle of the spotlight to do our calculations for the shader, and they are set at default values, as can be seen in code snippet 4.1.

Code 4.1: Shader dependencies

```
1  _MainTex ("Background Texture", 2D) = "white" {}
2
3  _DecalTex ("Hidden Texture", 2D) = "black" {}
4
5  _BumpMap ("Normalmap", 2D) = "bump" {}
6
7  _LightDirection("Light Direction", Vector) = (0,0,1,0)
8
9  _LightPosition("Light Position", Vector) = (0,0,0,0)
10
11 _LightAngle("Light Angle", Range(0,180)) = 45
12
13 _IntensityScalar("Intensity", Float) = 50
```

First a directional vector is created from the light position to pixel on the surface where the shader is attached. It is normalized in order to give it a magnitude of 1. This is important because next we need to find the dot product between this direction vector and the light direction variable, which has a magnitude of 1. These operations can be seen in code snippet 4.2.

Code 4.2: Dot product

```
1  float3 direction = normalize(IN.worldPos - _LightPosition);
2
3  float scale = dot(direction, _LightDirection);
```

Next we find the intensity of which to illuminate the pixel by. We subtract the cosine of half the cone radius in degrees from the dot product in order to find

which pixels are in the light cone and which ones are outside. The pixels outside of the cone will get a negative value. This can be seen in code snippet 4.3.

Code 4.3: Intensity

```
1 float intensity = scale - cos((_LightAngle / 2) * (3.14 / 180.0));
2
3 intensity = min(max(intensity * _IntensityScalar, 0), 1);
```

From this we clean up the values so all values are between 0 and 1 to use in the illumination of the pixels; all pixels within the light cone are 1, the pixels outside of it are 0, and the pixels in the transition area goes from 1 to 0, as can be seen in code snippet 4.3. And lastly the intensity is multiplied onto the alpha value in order to show or hide pixels from the surface. We add the color value to the decal and in the Albedo to tint the surface and light to the corresponding color from the light source. This can be seen in code snippet 4.4.

Code 4.4: Apply values to Albedo

```
1  float2 uv_hidden = IN.uv_DecalTex;
2
3  fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
4
5  half4 decal = tex2D(_DecalTex, IN.uv_DecalTex) * _Color;
6
7  o.Albedo = (decal.rgb * decal.a * intensity * 2) + c.rgb * (1 - intensity)
       + (c.rgb * _Color * intensity);
8
9  o.Emission = decal.rgb * decal.a * intensity;
10
11 o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
12
13 o.Alpha = intensity * c.a ;
```

4.1.2.2 Unity script

On the light source, which is a Unity spotlight, a script is attached to control the lighting and sending information to the shader in order to manage the the puzzles in the desired way.

First, since there are three different colours that effects the shader, an activeLightID is set to keep track of which light is turned on. At startup all lights are off, and therefore the intensity of the spotlight is also set to 0.

The Vive controller used as the flashlight in the game is instantiated to track if the trackpad is pressed and in which position it is pressed, as can be seen on code snippet 4.5.

Code 4.5: Instantiation

```
void Start()
{
  activeLightID = -1;
  lightSource.intensity = 0;
  trackedObj = GetComponentInParent<SteamVR_TrackedObject>();
  controller = GetComponentInParent<SteamVR_TrackedController>();
  controller.PadClicked += Controller_BtnPress;
  reveal.SetFloat("_IntensityScalar", lightSource.intensity);
}
```

After the trackpad is pressed we get the y-axis and the x-axis to sort the press into 1 of 4 different zones. Each zone corresponds with a colour of the light except one, which is reserved to turn off the light.

When the zone activated is determined, the activeLightID is set, the intensity is raised to 5 and the colour of the light is set.

Lastly the DecalTex of the desired stencil is sent to the shader along with the intensity. This is done in code snippet 4.6.

Code 4.6: Trackpad Press

```
private void Controller_BtnPress(object sender, ClickedEventArgs e)
{
  if (device.GetAxis().x != 0 || device.GetAxis().y != 0)
  {

    btnCoords = new Vector2(device.GetAxis().x, device.GetAxis().y);

    if (btnCoords.x < 0 && btnCoords.y > 0)
    {
      activeLightID = 1;
      lightSource.intensity = 5f;
      lightSource.color = Color.green;
      reveal.SetTexture("_DecalTex", greenTexture);
      reveal.SetFloat("_IntensityScalar", lightSource.intensity);
    }

    else if (btnCoords.x < 0 && btnCoords.y < 0)
    {
      activeLightID = 0;
      lightSource.intensity = 5f;
```

```
21        lightSource.color = Color.red;
22        reveal.SetTexture("_DecalTex", redTexture);
23        reveal.SetFloat("_IntensityScalar", lightSource.intensity);
24      }
25
26      else if (btnCoords.x > 0 && btnCoords.y > 0)
27      {
28        activeLightID = 2;
29        lightSource.intensity = 5f;
30        lightSource.color = Color.blue;
31        reveal.SetTexture("_DecalTex", blueTexture);
32        reveal.SetFloat("_IntensityScalar", lightSource.intensity);
33      }
34
35      else if (btnCoords.x > 0 && btnCoords.y < 0)
36      {
37        activeLightID = -1;
38        lightSource.intensity = 0;
39        reveal.SetFloat("_IntensityScalar", lightSource.intensity);
40      }
41    }
42  }
```

The last part of the script is sending the remaining values to the shader, that is needed to show the effect. These being the position, direction, angle and colour of the light source, as seen in code snippet 4.7.

Code 4.7: Sending values to shader

```
1  void Update()
2  {
3    device = SteamVR_Controller.Input((int)trackedObj.index);
4
5    reveal.SetVector("_LightPosition", lightSource.transform.position);
6    reveal.SetVector("_LightDirection", lightSource.transform.forward);
7    reveal.SetFloat("_LightAngle", lightSource.spotAngle);
8    reveal.SetColor("_Color", lightSource.color);
9  }
```

4.2 Riddle

The riddle is a simple puzzle: a short riddle is visible on one of the walls, and the user has to guess the answer for that. The answer will serve as a clue for the next puzzle, the Cipher wheel.

4.2.1 Design

The riddle did not change much from the time of its conception. Originally it was meant to have two parts, where the parts would be overlain on top of each other, but only one of them would be visible at a time. By using the flashlight to light a specific colour on the wall, the corresponding part of the puzzle would be visible.

The first part would have been the riddle itself, written out in text, and the second part would have consisted of symbols on the wall, and the user would have been prompted to choose the symbol that matched the answer of a riddle to progress in the room.

This version of the puzzle used the flashlight, but since the third puzzle that had been selected for the project uses a similar design as the riddle did, minor changes have been done in order to make the riddle different from the Light sphere and make it fit a little better with the rest of the puzzles. The second part has been scrapped and the riddle is now tied closely to the Cipher wheel. The riddle used in this product reads: "I do not have eyes, but once I did. Once I had thoughts, but now I am white and empty". The answer to the riddle is the word "Skull", and that word is then used as part of the key that is used to decrypt the second puzzle. More on this in the section 4.3.

4.2.1.1 Virtual design

The riddle is simple text on the wall with three possible answers to help the user. The flashlight is used to reveal the text as it is not visible to begin with. Different colours of the flashlight will reveal different parts of the riddle. The answer ("skull") is used as one part of the key used to solve the cipher wheel. The design of the full riddle can be seen on Figure 4.2, and the split up version that users will experience in the game can be seen on Figure 4.3

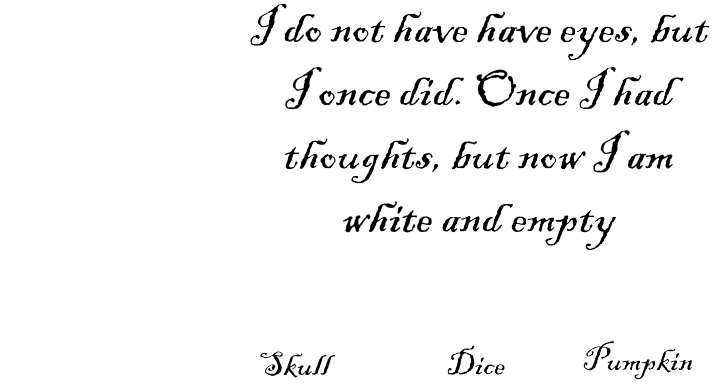
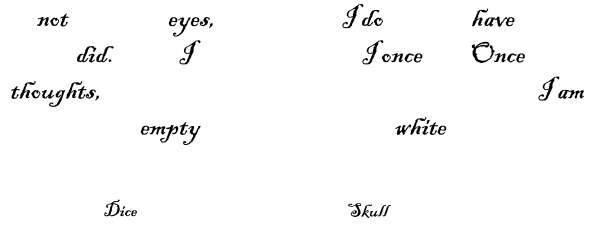
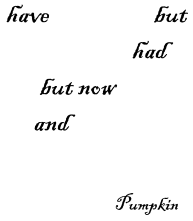


Figure 4.2: The riddle as seen in VR.



(a) The part of the riddle revealed by the blue light.

(b) The part of the riddle revealed by the green light.



(c) The part of the riddle revealed by the red light.

Figure 4.3: The three parts of the riddle after being split.

4.3 Cipher wheel

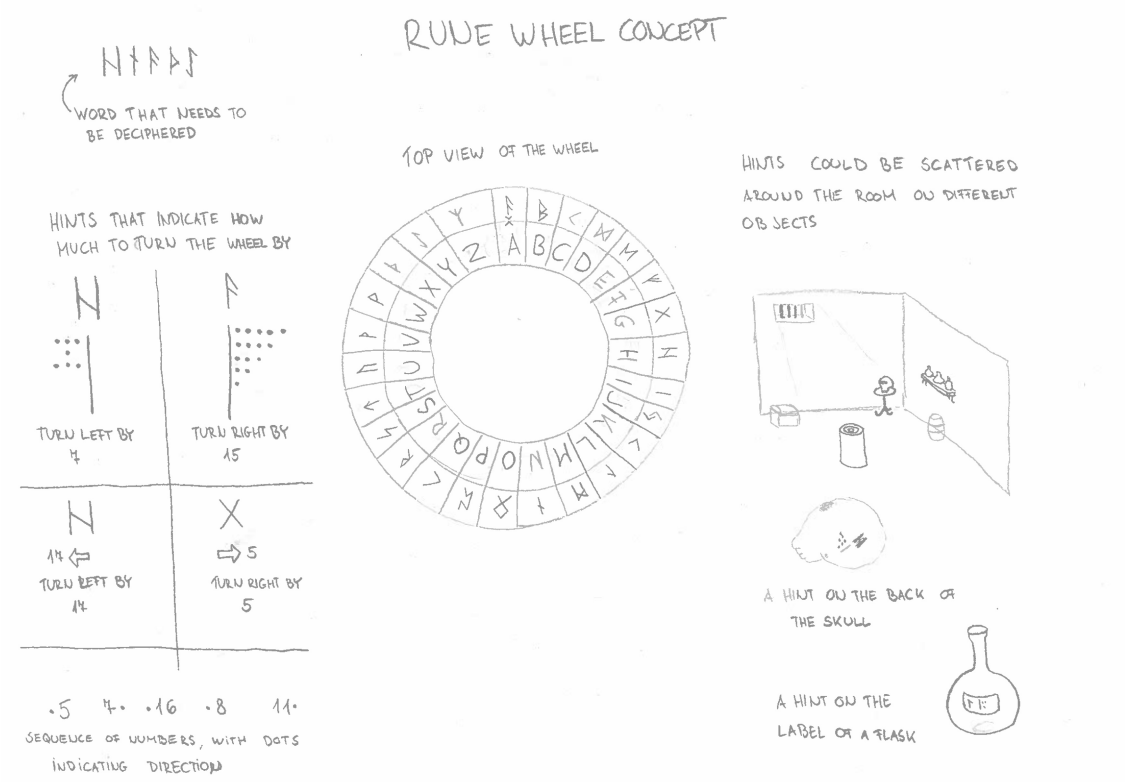


Figure 4.4: The initial concept for the cipher wheel.

The Cipher wheel is a puzzle based on the Alberti cipher wheel that consists of two wheels with various sets of characters on them that is used to decipher codes, as seen on Figure 4.4 In our project the code that is used for decrypting is the answer from the previous puzzle mentioned in section 4.2 and gives another word as a result. When the user solves this puzzle, they can move on to the next room. The final word is not used for any further puzzles.

4.3.1 Design

This puzzle remained unchanged since it was added to the list of puzzles. In our iteration, the character sets are the English alphabet on the inner wheel and a rune alphabet on the outer wheel. The goal of this puzzle is to use the answer to the riddle and the set of runes on the wall to input the correct solution using the wheel and the buttons, see Figure 4.5. The user has to align the outer wheel to the inner one so that the runic "A" points at the first letter of "SKULL", the answer to the

riddle, see Figure 4.6a. After the wheel has been aligned, the user then notes down which English letter the first runic letter on the wall corresponds to. Then the user presses the left button which switches the control from the runic ring to the frame, and turns the frame to the first rune from the sequence on the wall, see Figure 4.6b. Finally, the user uses the right button to input that letter, which is "B" in this case, see Figure 4.6c. After the user has gone through all the letters, they should be left with the word "BRAIN", which is the solution to the puzzle.

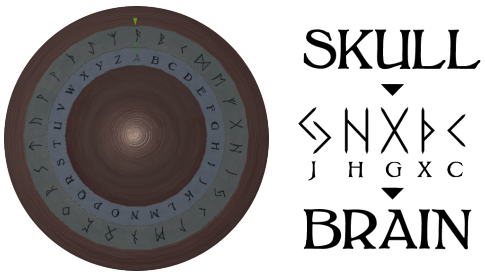
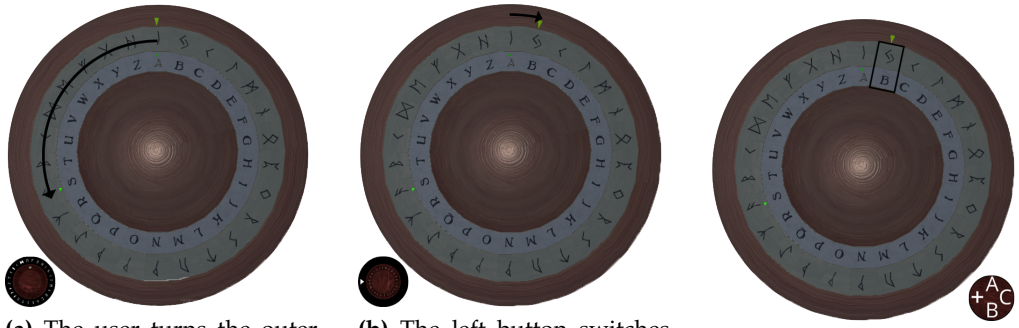


Figure 4.5: The cipher wheel in base position. We have the answer to the riddle: "SKULL", and the set of runes on the wall. In base position, the runes correspond to JHGXC, but when the outer ring is aligned correctly to each position, they will spell out "BRAIN" in the end, which is the solution to this puzzle.



(a) The user turns the outer wheel so that the runic "a" points to the first letter of skull, "S".

(b) The left button switches control to the frame, which is then aligned to the correct rune.

(c) The user then presses the right button, inputting the first letter of the solution, "B"

Figure 4.6: Steps of solving the cipher wheel. Steps a-c is repeated until the answer is input

4.3.1.1 Virtual design

The outer wheel was made using a pipe primitive with 26 subdivisions (1 for each letter of the alphabets), and some extra edge loops were added and modified at the border of the subdivisions so that a small groove would be present to help distinguish the subdivisions from one another. All but one subdivision were deleted, so

only one subdivision was being remodeled to ensure symmetry. The finished subdivision was then duplicated and rotated around the center pivot by 13,84 degrees and then combined with the previous subdivision to a single object consisting of 2 subdivisions. This was done continuously until all the subdivisions had been combined and formed the full wheel. One side was raised to give it a little inclination. Then this wheel was duplicated and scaled down to fit in the space enclosed by it. The dome was made by combining part of a sphere with a cylinder, both having 26 subdivisions to match the wheels, and finally the frame was made from a pipe primitive. The bottom of the wheel and the dome objects are aligned vertically, and the inner edge loop of the frame pipe has been bridged at the same height. The outer edge loop has been bridged somewhat lower. The model was exported as an FBX file.

Substance Painter was used for texturing of the model. The model consisted of 7 different objects, 3 arrows, the frame, the two wheels and the dome in the middle. Every object had a renamed lambert material assigned to them to make texturing with Substance Painter easier. The dome's material was named "dome", the inner wheel's was named "innerWheel", and so on.

The same wood material was assigned to the dome and the frame. The settings in Substance Painter were modified until a desired outcome closely resembling a wooden object was achieved. For the two wheels with the characters on them, two different bone material were chosen. The bone material, being light-coloured, allowed the characters to be legible and made the wheels stand out from the wooden parts. Different materials were used to emphasize the difference between the two wheels.

The two sets of characters have been made as a PNG with Adobe Photoshop, and those images were used as a stencil to paint the alphabet on the wheels. The textures have been exported to be used in Unity. The finished model can be seen in Figure 4.7.

Figure 4.7: The model of the cipher wheel as seen in VR during the pilot test

After running a pilot test with the cipher wheel, it was revealed that the puzzle

took too long. To lessen it's difficulty the runes were removed and the final design can be seen in Figure 4.8. This design does not involve the runes. The user now has to write out the letters of the answer to the riddle instead.

Figure 4.8: The model of the cipher wheel used in the final testing

4.3.1.2 Physical design

There was no exact replication of the cipher wheel, but a wooden pedestal was assembled with a cylinder with a diameter of 31cm and a height of 6cm. The pedestal is 85 centimeters tall, the cylinder for the cipher wheel is 6 centimeters tall, but is sunk 1 centimeter into the top of it, making the object 90 centimeters tall at its tallest point.

There are two other cylinders, both with a diameter of 5cm and a height of 7cm, representing the buttons that are used to switch which part is being rotated, and to lock the chosen letter as part of the final answer.

The pedestal's model was recreated in Maya, and has been combined with the model for the cipher wheel.

4.3.2 Implementation

In the hierarchy, the RuneWheel object is split into a dome, an inner ring, a middle ring and a frame, with a shared empty game object as a parent. The script is attached to this parent object. The script contains a reference to the middle ring and the frame of the cipher wheel, as well as to a Vive tracker. Additional references are made to sound files, a door object, point lights and a public PassCode struct, where the solution of the puzzle can be set. The rest of the variables are there to help with the calculation and tracking of the puzzle.

When the game is run in "virtual mode", without physical objects, the orientation of the rings can be changed by grabbing them with the Vive controller and rotating them. This was done by attaching SteamVR's Circular Drive script to the rings. Inputting a letter is done by hovering the controller near or in the right button on the pedestal and pressing the trigger button on it.

In "physical mode" where physical objects are included in the game, a given ring's orientation can be changed by grabbing the pedestal and rotating it by hand, where the tracker's rotation on top of the pedestal's wheel sets the ring's rotation. Changing which ring we want to rotate can be done by pressing the left button on the pedestal, when the user does this, a snapshot of the tracker's current rotation is saved, as well as the ring's current rotation, and the rotation of the ring that is switched to is calculated by the tracker's rotation, minus the snapshot, plus the saved rotation. This ensures that the rings' orientation doesn't jump around when the player switches between them. The script for this can be seen in code snippet 4.8.

Code 4.8: Calculating the rotation in the Update() method and switching rings in Tracker mode.

```
1       ...
2       if (isTrackerVersion)
3         {
4             rings[BoolToInt(middleRingActive)].rotation =
                  Quaternion.Euler(0, tracker.eulerAngles.z - rotationSnapshot
                  + savedRotations[BoolToInt(middleRingActive)], 0);
5         }
6        ...
7
8    void ChangeWheel()
9     {
10        rotationSnapshot = tracker.eulerAngles.z;
11        savedRotations[BoolToInt(middleRingActive)] =
              rings[BoolToInt(middleRingActive)].transform.eulerAngles.y;
12        leftButton.GetComponent<Renderer>().material =
              materials[BoolToInt(middleRingActive)];
13        middleRingActive = !middleRingActive;
14    }
```

Inputting a letter is done by pressing the right button. Since the solution to the puzzle is five combinations of letters/runes in succession, it was decided that by calculating a number from 0 to 25 from the rings' orientation is the best way to represent them. Storing the solution therefore is done in an array of PassCode structs, which have two integers as variables, effectively an array of 2-length arrays, however this way they could be serialized in the inspector, and naming was more straightforward. The displacement values are calculated with the equation 4.1,

with the condition that if the displacement is calculated to 26, it will be changed to 0. This had to be done because between 0 and -6.92 degrees, the equation would calculate 26 and would cause an array overflow error.

$$displacement = \left\lfloor \frac{26 \cdot (Rotation_{ring} + \frac{1}{2} \cdot \frac{360}{26})}{360} \right\rfloor \tag{4.1}$$

The code for the displacement values is in code snippet 4.9.

Code 4.9: Calculating the rings' displacement in the Update() method

```
...
middleDisplacement = Mathf.FloorToInt(26 * (rings[1].eulerAngles.y +
    increments / 2f) / 360f);
 if (middleDisplacement == 26) middleDisplacement = 0;
 outerDisplacement = Mathf.FloorToInt(26 * (rings[0].eulerAngles.y +
     increments / 2f) / 360f);
 if (outerDisplacement == 26) outerDisplacement = 0;
 ...
```

The displacement values are checked against the values of the PassCode array, starting from 0 to 4. The array index is set by the progress variable, so the program knows where we are with the puzzle. Whenever a correct rune-letter combination is entered, a light is turned on, and the progress variable is advanced. The code for checking the conditions can be seen in code snippet 4.10. When the final correct letter is entered, the door of the room opens, and the puzzle stops taking in any more inputs.

Code 4.10: Condition check methods for the RuneWheel

```
bool CheckCondition(int conditionIndex)
 {
     return (middleDisplacement == passcode[conditionIndex].middleSymbol
         && outerDisplacement == passcode[conditionIndex].outerSymbol);
 }

 public void CheckConditionOnButtonPress()
 {
     if (CheckCondition(progress))
     {
         Debug.Log("right");
         progress++;
         lights[progress - 1].SetActive(true);
         soundSource.PlayOneShot(lightBulbSound, 1.0f);
     }
     else
```

```
16          {
17              Debug.Log("wrong");
18          }
19      }
```

4.4 Light sphere

The light sphere was chosen as it shared several similarities with the cipher wheel. The light sphere uses a rotating half-sphere of glass to project parts of an image onto a wall. Different parts of the image is revealed and can turn depending on the colour of the light being shined on it. Because both the light sphere and the cipher wheel utilizes a rotating wheel, the same physical object could be used for both puzzles, which was necessary to implement two rooms in the same physical space.

4.4.1 Design

Initially the light sphere projected a texture of scattered letters across the walls, which changed depending on light. The idea was that this should reveal the riddle explained in section 4.2, but because the cipher wheel used the same physical object, the two puzzles had to be in different rooms. This idea can be seen in the concept on Figure 4.9.

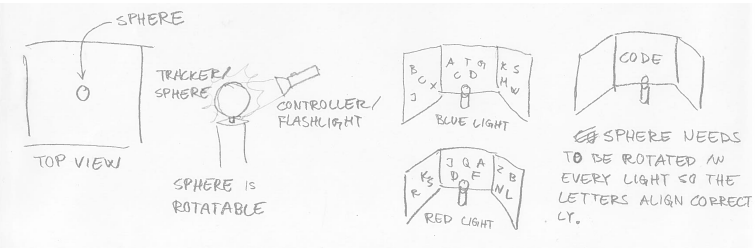


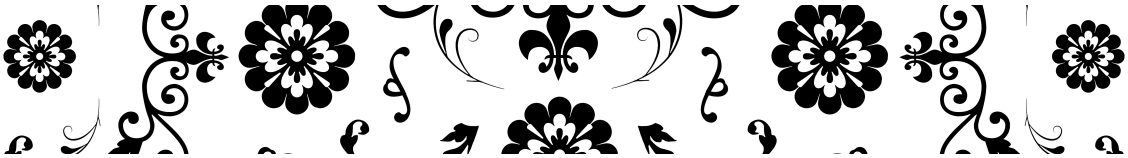
Figure 4.9: The concept of the light sphere.

Instead the projected textures was changed into a picture which was split into three bars. By turning each bar to a specific spot the combined image would be revealed and allow the user to progress to the next step of the game. These bars can be seen in Figure 4.10.

4.4.1.1 Virtual design

The Light sphere's model was closely based on the cipher wheel's. The reason behind this was for both of them to have the same physical dimensions, so that the same physical object could be used for the two. If the Light sphere was bigger

(a) The part of the image revealed by the red light.



(b) The part of the image revealed by the blue light.

(c) The part of the image revealed by the green light.

Figure 4.10: The different parts of the image in the light sphere puzzle.

than the cipher wheel, for example, with the inclusion of an actual sphere, the risk would have been for the user to try and grab it even though there was nothing there in the real world. Thus, the light sphere is closer to a light lens, which can be seen in Figure 4.11.

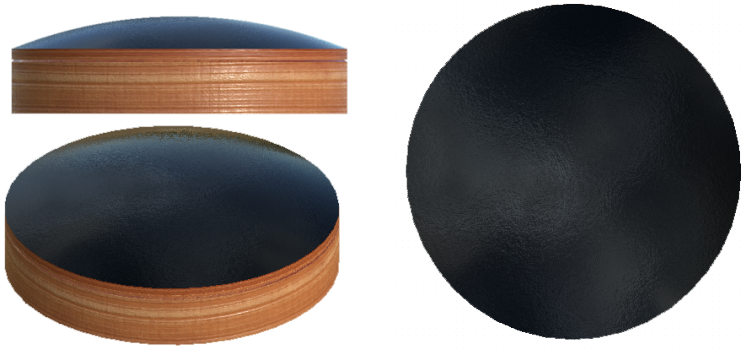


Figure 4.11: The model of the light sphere as seen in VR.

4.4.1.2 Physical design

The light sphere uses the same physical pedestal as the cipher wheel, so that both puzzles can appear in two different virtual rooms. To see the design process of the physical pedestal, see section 4.3.1.1.

4.4.2 Implementation

The Light Sphere puzzle's implementation was done in two parts. First the puzzle, and the logic behind it, then an additional script on the flashlight, so that the puzzle only activates when the user shines a light on it.

The puzzle itself is one object with an empty game object as a parent in the hierarchy, with the script attached to the parent. The script has a reference to a Vive controller to take in inputs from the flashlight for changing colours, an array for the projectors and a reference to the sphere's game object. Additional objects are referenced in the script, for example audio sources and a chest object, which will be unlocked when the puzzle is solved. The solution of the puzzle is three independent rotations, one for each light, so they can be stored in a float array.

When a given light is active, the respective projector's orientation is calculated by the sphere's orientation minus the correct rotation for the light, this way when all three projectors are in the correct orientation, they all face the same direction. When the user changes the flashlight's colour, the rotation of the projector they switch to is loaded to the state it was when they switched off of it. This is achieved in two different ways. In "virtual mode", without the trackers, whenever the user switches from a colour to another, the rotation of the sphere is saved, and a previously saved rotation is loaded in. In "physical mode", the sphere's orientation has to be calculated from multiple values, because the tracker's rotation is continuously updated. When the rotation changes, a snapshot of the tracker's current rotation is saved, and the sphere's rotation is then calculated by the equation seen in equation 4.2.

$$Rotation_{sphere} = Rotation_{tracker} - Snapshot_{tracker} + savedRotation_{currentLight} \quad (4.2)$$

One challenge that had to be overcome in "tracker mode" was that the orientation of the first projection was always off. That was because when the puzzle starts up, both the saved rotations and the rotation snapshot values are zero, thus the sphere's rotation is initially equal to the tracker's rotation. This issue was circumvented by having a two second calibration phase where the tracker's rotation is saved into the rotation snapshot, as seen in code snippet 4.11. The reason this is not done in the Start() or Awake() methods is because the tracker is inaccurate in the first few frames after startup.

Code 4.11: Calibrating the initial rotation in the Update() method

```
1 if (isTrackerVersion && elapsedTime <= calibrationTime)
2 {
3     transform.position = new Vector3(tracker.position.x,
          transform.position.y, tracker.position.z);
4     elapsedTime += Time.deltaTime;
5     rotationSnaphot = tracker.eulerAngles.z;
```

```
6 }
```

When an answer is submitted, the program checks the sphere's current rotation as well as the other two saved rotations against the conditions. Submitting an answer in the virtual scenario is done by touching the right button on the pedestal with the controller and pressing the trigger, and in the physical scenario the user has to press the right button with their hand. The solution does not have to be precise to the degree, it is done so that it has to be between a given angle around the solution value. When the correct answer is entered, all three projections light up, showing the entire image as one, with a slight imperfection due to the previously mentioned threshold. Then the program calls the chest's UnlockChest() method, and turns on a spotlight shining on the chest, indicating that it is now interactable. Additionally, the sphere stops taking in new inputs, be it rotation or light colour changing. This can be seen in code snippet 4.12.

Code 4.12: What happens when the puzzle is solved

```
else if (!chestOpen)
{
   for (int i = 0; i < 3; i++)
       {
          lights[i].SetActive(true);
       }
   winLight.SetActive(true);
   chest.UnlockChest();
   chestOpen = true;
   rotationSnaphot = tracker.eulerAngles.z;
   sphere.localRotation = Quaternion.Euler(0, rotationSnaphot, 0);
}
```

After the logic of the puzzle was done, A solution was implemented so that the sphere would only light up if the user shines the flashlight on it. The script for it was attached on the flashlight's parent object. Originally it was coded using a cone shaped collider fit to the size of the cone of the flashlight's spotlight part, hence the script's name "LightCone.cs", however issues have come up when multiple colliders were hit at the same time. This was fixed by changing the cone collider into a simple raycast projecting from the front of the flashlight, and set the raycast to ignore everything but the light sphere using a layermask. Of course this is not as accurate as the cone collider, but for the purpose of testing it was sufficient. The way it works is, when the raycast detects an object with a "Light Sphere" component, it calls its TurnLightOn() method, and saves the component. When the raycast doesn't hit anything, it checks if there already is a saved "Light Sphere", and if there is, it calls its TurnLightOff() method. If the flashlight is turned off, then the raycast doesn't happen, and instead it checks for a saved "Light Sphere" and

turns it off if it exists. Doing it this way is possible because there is only one Light Sphere puzzle in the scene, and if it hasn't been saved already, then it is already turned off. The exact implementation can be seen in code snippet 4.13.

Code 4.13: The Update() method of the LightCone.cs script

```
1  void Update()
2  {
3      if (LRSource.activeLightID != -1)
4      {
5          Ray ray = new Ray(transform.position, transform.forward);
6          RaycastHit hitInfo;
7          int sphereMask = LayerMask.GetMask("Sphere");
8          if (Physics.Raycast(ray, out hitInfo, 100, sphereMask))
9          {
10             Debug.DrawLine(ray.origin, hitInfo.point, Color.red);
11             ls = hitInfo.collider.GetComponentInParent<LightSphere>();
12             ls.TurnLightOn();
13         }
14         else
15         {
16             Debug.DrawLine(ray.origin, ray.origin + ray.direction * 100,
                   Color.green);
17             if (ls != null) ls.TurnLightOff();
18         }
19     }
20     else if (ls != null) ls.TurnLightOff();
21 }
```

5. Additional assets

This chapter will cover all the virtual assests created for the project that were not covered in chapter 4.

5.1 Pedestal

This section documents the process of designing the pedestal.

5.1.1 Design

The pedestal is a wooden stand that was created to hold 3D objects used in the puzzles and enable users to interact with those virtual objects physically.

5.1.1.1 Virtual design

In order to make the model as accurate to its real-world counterpart as possible, a photo was taken of the pedestal and it was set up via an image plane in Maya to serve as reference. The column was shaped out of a cylinder that was elongated to match the size correctly. The details on the column were achieved by adding edge loops along the cylinder and then scaling them to the correct size. Some edges were beveled to make them smoother. The bottom element was shaped out of a polygon cube, scaled correctly, and the rounded edges were achieved with using the bevel tool. The top element was created similarly, but the edges were not rounded. The three cylinders were created from rescaled cylinder primitives. The holes in the top element were not created, so the cylinders were scaled taking this and the height of the cipher wheel and the Light sphere into account.

Texturing was done with Substance Painter. The pedestal consists of 6 different objects: the top and bottom element, the column, and the three cylinders. All but the largest cylinders share the same material, and the largest cylinder shares the material of the cipher wheel and Light sphere.

Figure 5.1: The model of the pedestal as seen in VR

5.1.1.2 Physical design

The pedestal was made to house a rotating cylinder and give space to two buttons. The pedestal was meant to be used standing up, and the height was chosen to match the size of an average kitchen counter (IKEA's counters are 88 centimeters excluding the cover panels which are between 1-2 centimeters). The buttons had to be big enough to be easily accessed while wearing the headset, and the smaller the button, the more precise its placement would have to be. The width of the button was chosen to be about the size of an average palm, which various sources[4, 5, 6] say to be around 7-7.5 centimeters. The top element was made to be wide enough for the buttons in the corners could be reached comfortably, and the size of the large cylinder was chosen so there would be enough space between its and the top element's edges for the user's arm when they reach for the button.

Figure 5.2: The physical version of the pedestal

5.1.2 Implementation

The pedestals for the cipher wheel and the light sphere puzzles were separate prefabs, with a small script attached to the right button of each. The scripts were similar in functionality, but for ease of implementation they were separated. The scripts were responsible for interactability in the virtual scenario. They had a reference to a Vive controller, a puzzle, and they had a boolean to signify if a collider has entered the button or not. The method checking the puzzle's answers was subscribed to the controller's trigger click event, and only ran if a collider was inside the button. The implementation for the cipher wheel can be seen in code snippet 5.1. The implementation for the light sphere is near identical.

Code 5.1: The button press method in RWTrigger.cs

```
1 void RWbtnPressed (object sender, ClickedEventArgs e)
2 {
3     Debug.Log(inCollider);
4     if (inCollider)
5     {
6         rwPuzzle.CheckConditionOnButtonPress();
7     }
8 }
```

Since interacting with physical objects in VR is an essential part of this project, a method of integrating the actions done with the physical objects needs to be implemented. Specifically, the implementation of two physical buttons are needed, so in-game actions can be triggered with the push of a button. To achieve this, an Arduino **??** is used to listen for button presses and feeding the data into Unity **??**.

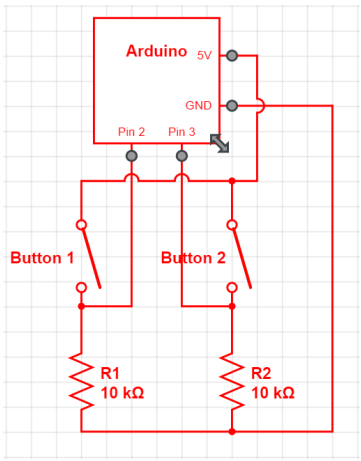
5.1.3 Physical Setup

The physical setup of the two buttons is just the two buttons that complete the circuit between 5V supply and the measured pins (2 and 3). A pull-down resistor is added to ensure that the pins always measure 0V when the switch is open. The circuit diagram can be seen in Figure 5.3a.

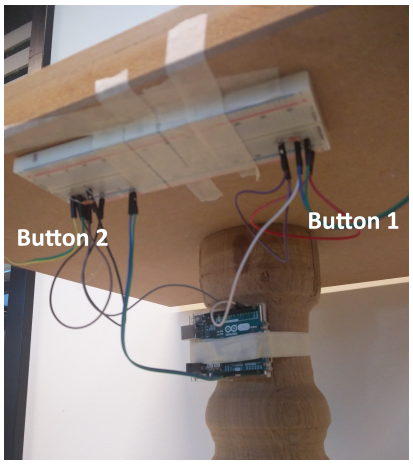
5.1.4 Arduino Implementation

As shown in section 5.1.3 the buttons are connected to pin 2 and 3 on the Arduino. This means that the value (HIGH or LOW) on those pins are the information that Unity needs to know if the buttons are pressed. In Code 5.2 the function that sends the data to Unity over a serial connection is shown.

Code 5.2: Implementing the two buttons on the Arduino



(a) Circuit diagram showing the implementation of the two buttons



(b) The physical setup of the buttons.

Figure 5.3: The setup of the buttons with the Arduino on the physical model of the pedestal.

```
1  void SerialOutput() {
2    //Time to output new data?
3    if(millis() - serialLastOutput < serialOutputInterval)
4      return;
5
6    serialLastOutput = millis();
7    buttonOne = digitalRead(BTN_ONE_PIN);
8    buttonTwo = digitalRead(BTN_TWO_PIN);
9
10   //Write data package to Unity
11   Serial.write(StartFlag);    //Flag to indicate start of data package
12   Serial.print(millis());     //Write the current "time"
13   Serial.print(Delimiter);    //Delimiter used to split values
14   Serial.print(buttonOne);       //Write a value
15   Serial.print(Delimiter);    //Write delimiter
16   Serial.print(buttonTwo);       //...
17   Serial.println();           // Write endflag '\n' to indicate end of package
18 }
```

The function checks if it is time to send new data to Unity in line 3-4. If it is time to send new data, the values on the buttonpins are saved in line 7 and 8. To signal Unity that a data package is coming in, a start flag is sent along with the runtime of the Arduino, followed by the defined delimiter. Then the values of buttonOne and buttonTwo is sent in line 14 and 16, seperated by the delimiter (line 15). The message ends with a newline, to tell Unity that no more data is coming in this data package.

5.1.5 Unity script

When the data is received in Unity it is saved in a string. The values are then split into an array using the delimiter. The value from the buttons are 0 when not pressed and 1 when pressed. These values are converted into a boolean using the StringToBoolean function, that simply returns true if the values are 1 and false otherwise.

Code 5.3: Implementing the two buttons on the Arduino

```
1  //Event handler
2  public delegate void NewDataEventHandler(Arduino arduino);
3  public static event NewDataEventHandler NewDataEvent;
4
5  ...
6
7  private void ProcessInputFromArduino(string serialInput)
8  {
9    ...
10
11   string[] values = serialInput.Split('\t'); //Split the string between the
         chosen delimiter (tab)
12   ArduinoMillis = uint.Parse(values[0]);  //Pass the first value to an
         unsigned integer
13   ButtonOne = StringToBoolean(values[1]); //Get value from button on pin 2
14   ButtonTwo = StringToBoolean(values[2]); //Get value from button on pin 3
15
16   if (NewDataEvent != null) //Check that someone is actually subscribed to
         the event
17    NewDataEvent(this);   //Fire the event in case someone is
18 }
19
20 bool StringToBoolean(string str) {
21   if (int.Parse(str) == 1)
22       return true;
23   else
24       return false;
25 }
```

In line 2 and 3 an eventhandler is created, and in line 16 and 17 the subscriptions to the event is handled. This event fires every time new data comes into the Arduino, and enables Unity to react to incoming data. In Code 5.4 the incoming data is handled to fire events when a button is pressed.

Code 5.4: Implementing the two buttons on the Arduino

```
1  void NewData(Arduino arduino)
```

```
2     {
3         if (!btnOnePressed)
4         {
5             if (arduino.ButtonOne)
6             {
7                 if (BtnOnePress != null)
8                     BtnOnePress();
9                 btnOnePressed = true;
10            }
11        }
12        else
13        {
14            if (!arduino.ButtonOne)
15                btnOnePressed = false;
16        }
17
18        if (!btnTwoPressed)
19        {
20            if (arduino.ButtonTwo)
21            {
22                if (BtnTwoPress != null)
23                    BtnTwoPress();
24                btnTwoPressed = true;
25            }
26
27        }
28        else
29        {
30            if (!arduino.ButtonTwo)
31                btnTwoPressed = false;
32        }
33    }
```

This function is subscribed to the NewDataEvent, so every time new data arrives, it checks if any button is pressed. The first time a button returns true, an event fires for the respective button, and sets a boolean for the button to true. The boolean is used to check if the button is held down, and the button only fires the event once per press. When the button is released the boolean is reset to false, so the function is ready to fire the event again the next time the button is pressed. Other scripts can now subscribe to the event of each button, and execute some code whenever a button is pressed.

5.1.6 Audio

An audioclip is added to the button press events on the pedestal to achieve feedback for the user. In this case we added a switch sound.

5.2 Chest

5.2.1 Design

The chest is made of wood with a simple latch at the front. It serves as the container for the reward the player gets at the end of the test.

5.2.1.1 Virtual design

The chest was separated into two major parts: the bottom and the lid. The bottom was modeled from a cube primitive, which was extruded inward after being scaled to the right size. The lid was created similarly, with a beveled edge to make the top rounded. Two latches were created on the back were modeled from a cylinder and two cube primitives.

Substance painter was used to create the texture for the model. The parts that are painted black on the edges of the chest were made to look like metal bands, and the wooden parts were given additional detail with the use of stencils. These details include wear and handprints.

Figure 5.4: The chest as seen in VR

5.2.1.2 Physical design

The chest was not created specifically for the project. Its dimensions are 22.5 cm by 27 cm by 39.5 cm.

Figure 5.5: The physical version of the chest

5.2.2 Implementation

The chest is split into its lid and base in the hierarchy, and the Chest class is applied to these two objects' shared parent. The class, seen on code snippet 5.5, has a reference to the lid, the lid's collider and a Vive tracker. It also has two booleans, one two differentiate between the virtual and physical version and one to check if it is locked or not. It has one method, to unlock the chest. In the virtual scenario the unlock method enables the lid's collider to make it interactable, in the physical scenario it sets the lid's rotation to the tracker's rotation so it can be opened by opening the real chest in the room.

Code 5.5: The Update() method is responsible for the physical scenario, while UnlockChest() can be accessed by other classes to unlock the chest

```
1   void Update()
2    {
3        if (isTrackerVersion)
4        {
5            if (!isLocked)
6            {
7                lid.transform.localRotation =
                     Quaternion.Euler(-tracker.eulerAngles.x - 90, 0, 0);
8            }
```

```
9         }
10    }
11
12    public void UnlockChest()
13    {
14        isLocked = false;
15        lidCollider.enabled = true;
16    }
```

5.2.3 Audio

An audioclip is included on the chest for when it unlocks. This was to steer the players attention towards the chest at the end of the game.

5.3 Barrel

This section will explain the process of implementing the barrel.

5.3.1 Design

A problem was encountered when adding a physical chest as part of the second room, as it should not appear virtually in the first room. To make sure people would not walk into the chest, while they could not see it in the first room, a virtual barrel was added as a placeholder to deter them from walking in that spot.

5.3.1.1 Virtual design

The barrel's body was modeled from a cylinder primitive. One pipe primitive was rescaled around the middle of the body. This pipe was then duplicated, and moved towards the top of the barrel about halfway between the edge and the middle band, where it was rotated to match the curvature of the body. Once this was in place, this was duplicated again, and moved to the edge. It was rotated and rescaled. These top two bands were then duplicated, rotated a 180 degrees, and moved to the bottom half of the barrel. The five bands were then combined into a single object.

The barrel was rotated 90 degrees, and was scaled to match the length of the chest. The topmost edge of the barrel (an edge on the middle band) corresponded with the top edge of the arch in the lid of the barrel. The stand would take up the remaining space between the bottom of the barrel and the bottom of the chest.

A cube primitive as wide as the chest was created with four subdivisions along the length of it. The two edges at ¼ and ¾ were raised and beveled, while the middle edge was lowered and beveled. This gave a curve in the stand in which

the barrel would rest. After this all the edges were beveled with a smaller fraction to give a smoother look to the model. This object was aligned using snapping to grids, and it was duplicated and moved to the other end of the barrel, retaining symmetry. A cylinder was used to create the rods between the stands. After correct rescaling and positioning, it was duplicated and moved to the correct position on the other end of the stands. The rods and the stand were combined into a single object.

Texturing was done using Substance Painter. The bands were given a metallic material, the body got a more rugged wood material, and the stands were given a lacquered wood material.

Figure 5.6: The barrel as seen in VR

5.3.1.2 Physical design

This object has not physical counterpart. However, it will be in the same position in the virtual world as the chest is in the physical world.

5.4 Light bulb

5.4.1 Design

A light bulb was created to act as the source of the ambient light in the rooms.

5.4.1.1 Virtual design

The bulb was created by taking a sphere primitive, whose top few rows were elongated. Some edge loops were moved and scaled to give the bulb a smooth curve.

The coil in the middle was created from a torus primitive that had one quarter deleted. The ends were extended, and after that the item was duplicated and flipped. The two objects were combined into one object and placed accordingly within the bulb.

A blinn material was assigned to the glass component of the bulb, and with its transparency reduced, it gives the impression of a glass object.

Figure 5.7: The bulb as seen in VR

5.4.2 Audio

In the game the lightbulbs are used to signal when the user do a successful step in the first puzzle. Here an audioclip was added to draw attention to the lights when the user was looking down.

5.5 Door

5.5.1 Design

5.5.1.1 Virtual design

The wooden planks of the door were modeled from cube primitives. The cubes were scaled to size first. Following this, edge loops were added with the Insert Edge Loop tool along the height of the door. The planks were made to look more organic showing sign of rot and wear that would occur naturally by modifying the edge loops either by moving or rescaling them. Once the planks were in order, an object acting as a metal bar, modeled from a cube primitive was added along the width of the door. Five sphere primitives were added to the horizontal bars to act as bolts.

A hinge, consisting of multiple different objects were added to one side of the horizontal bars. The part that is attached to the wall was modeled from a cube primitive and has 4 spheres representing bolts. The part connecting the two objects of the hinge was modeled from a cylinder. More subdivisions were added. Those edges were then scaled and moved to achieve the desired form. The larger part of the hinge, which is attached to the horizontal bar, was created using a cube primitive with multiple subdivisions. The edges were scaled and combined to achieve the arrowhead-like protrusions and tip. The holes in this part were achieved by circularizing the desired area. The faces on the two sides were deleted, and the edge loops were bridged to close the model.

Figure 5.8: The door as seen in VR

5.5.1.2 Physical design

The door is only present in the virtual environment, an so there is no physical model of it.

5.5.2 Implementation

The door is split to the door itself and its hinges in the hierarchy. There is also a third, empty object acting as a pivot that the door can rotate around. The script is applied to a shared parent object, and is responsible for opening and closing the door. Parts of this script can be seen on code snippet 5.6. The ActivateDoor() method changes the door's current state, and can be called by other classes, which is utilized in the cipher wheel puzzle. Each door also has trigger zones for opening

and closing, by utilizing the PlayerTracker class, and modified methods from the RoomSwitcher, that ignore gaze direction and only switch based on where the player stands.

Code 5.6: Part of the Door's update method, checking for the first condition, and the OpenDoor() method, rotating the door around a pivot

```
1   ...
2   if(CheckCondition(openCondition) && !conditionFulfilled)
3   {
4       ActivateDoor();
5       conditionFulfilled = true;
6    }
7    ...
8
9   void OpenDoor()
10   {
11      if (isOpen) return;
12      board.transform.RotateAround(pivot.position, Vector3.up, 90);
13      isOpen = true;
14   }
```

5.6 Note with nail

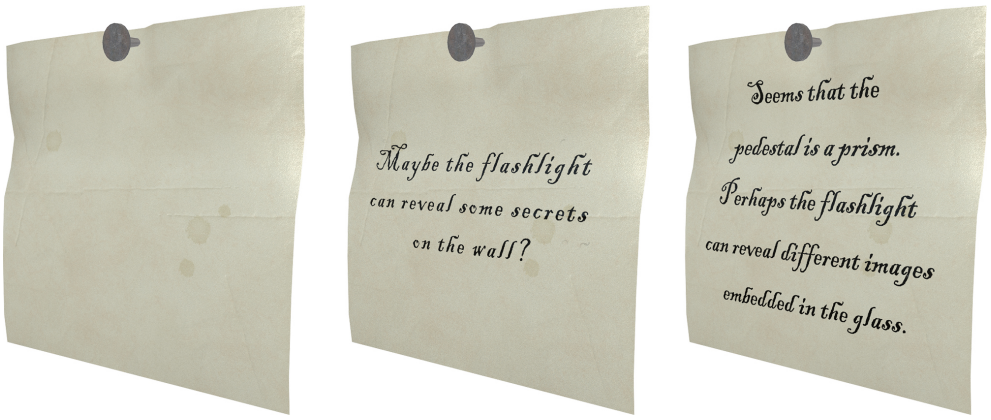


Figure 5.9: The note, kept up with a nail, as seen in VR with different hints to help the player

5.6.1 Design

The note served the purpose of helping the user getting started with the puzzles.

5.6.1.1 Virtual design

The paper was made from a plane with 10 subdivisions. Transformations were then applied to different edges to curl the paper as well as creasing it a bit where the nail was going to be. The nail was made from a cylinder. The cylinder had its bottom edges scaled in and extruded for the spiky part. These two models were both kept with a low amount of polygons since the paper did not need a lot of details and the nail would be too small for anyone to notice a difference.

5.6.1.2 Physical design

Neither the paper nor the nail had any physical representation of their models made.

5.7 Wall

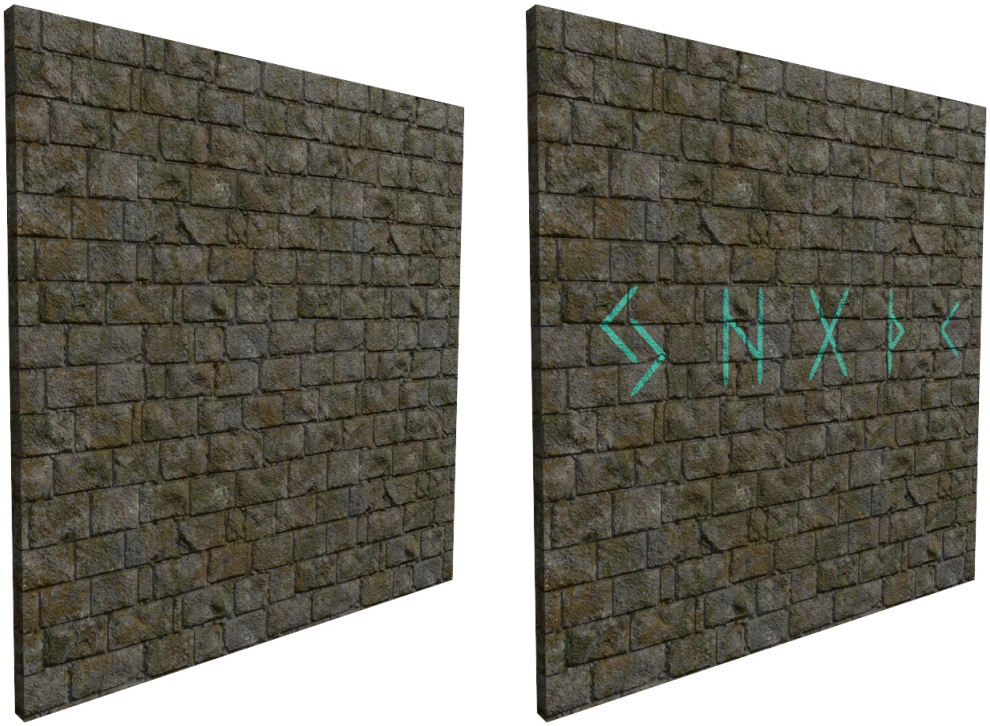


Figure 5.10: The wall to the left displays the texture used for most walls. The wall on the right shows the same texture but with the runes applied to it.

5.7.1 Design

The walls were implemented in the project to guide the user in where they could walk. Their model was textured to resemble castle walls with uneven stones and mossy growth.

5.7.1.1 Virtual design

As most of the details of the walls were made with the textures and their different maps the mesh itself was made as simple as possible. There were a total of 6 different walls made to fit different sizes. Each wall is made from a cube with 6 polygons to keep the amount polygons as few as possible.

5.8 Floor

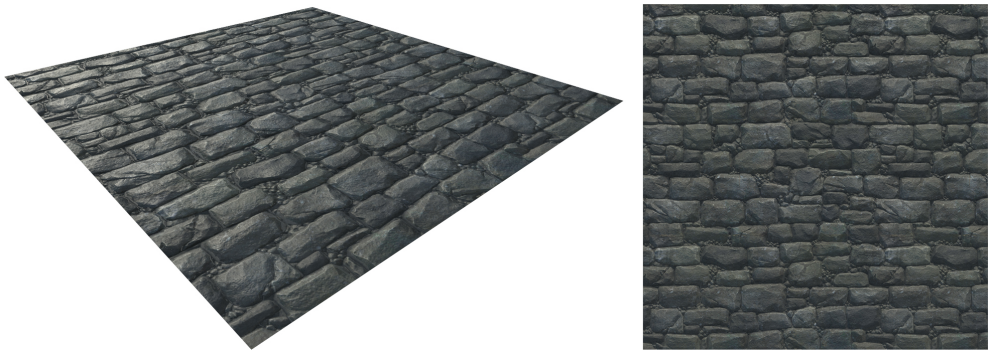


Figure 5.11: The plane used as the floor in the virtual environment.

5.8.1 Design

The floor was made to aesthetically fit with the rest of the room. A texture with cobblestones was used to aid the impression of the caste.

5.8.1.1 Virtual design

The floor was made with a flat plane, consisting of a single polygon, as the texture maps would add all the details.

5.9 Additional scripts

5.9.1 VR Position Helper

For every object that needed to be tracked in the physical scenario, a way had to be found so that their position and orientation matches their real life counterparts

in the game. For orientation, the physical objects were rotated before testing, and that was sufficient, but for the position, more precision was needed. A short script was created that, after the object is activated, tracks the position of the tracker for a few seconds, and then stops the tracking as seen in code snippet 5.7. The tracking could not be placed in the Awake() or the Start() methods, because the tracker is inaccurate in the first few frames, neither could the tracking be continuous, because inaccuracies in tracking the Vive tracker, or in the case of the puzzles, interacting with the pedestal could influence the objects' position and make them jitter and jump around. Since the trackers were in the middle of the objects, displacements in game did not have to be accounted for. The script was attached to both puzzles' pedestals, the chest and the barrel, with the barrel not having a physical counterpart, but using the chest's location as a base, so people in the physical scenario's first room wouldn't trip over "empty" space.

Code 5.7: The Update() method of the VR Position Helper script

```
1 void Update()
2 {
3     if (isTrackerMode && elapsedTime <= calibrationTime)
4     {
5         transform.position = new Vector3(tracker.position.x,
              transform.position.y, tracker.position.z);
6         elapsedTime += Time.deltaTime;
7     }
8 }
```

6. Room design

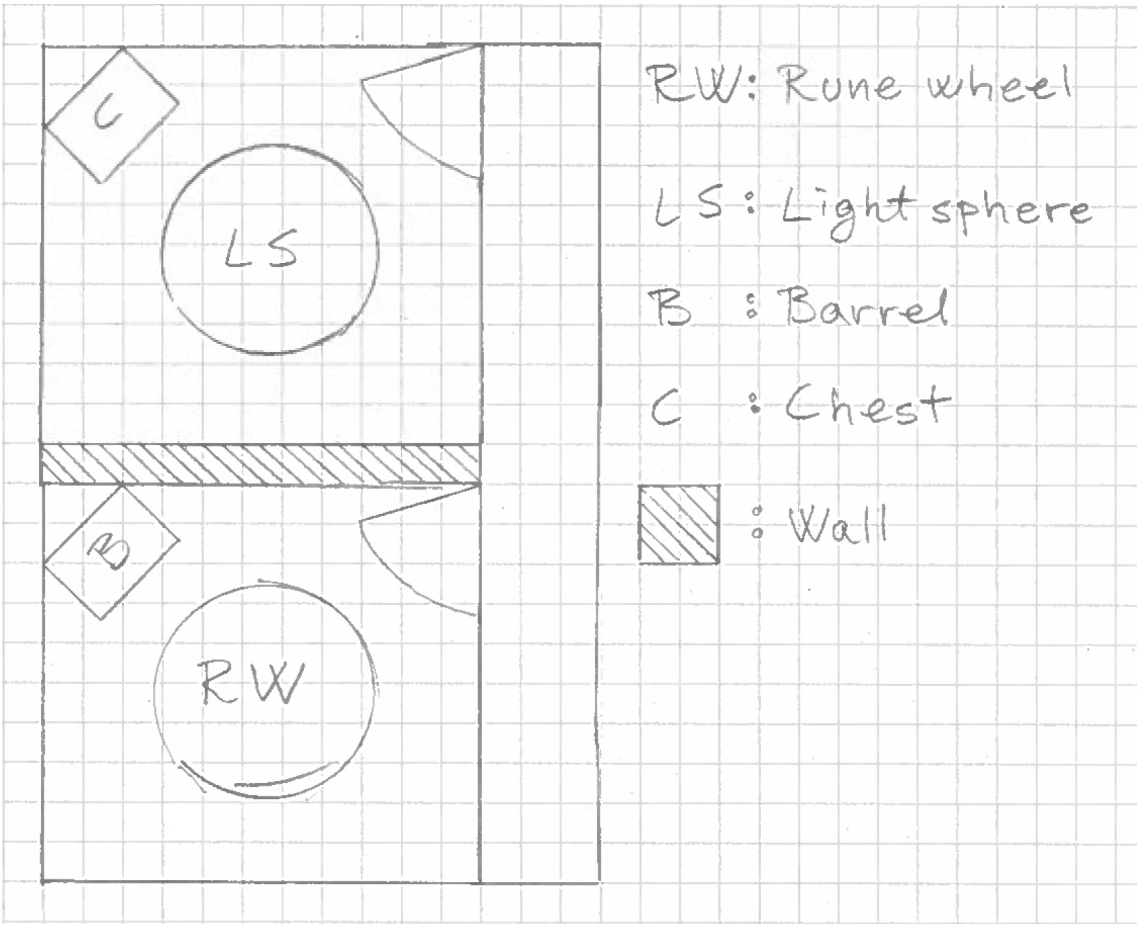


Figure 6.1: An initial sketch of the room layout

Once the design of all the puzzles were done, the next step was to design the layout of the rooms. The room layout was worked out to ensure that the trackers would not change position in the different rooms. The sketch of the layout, as seen in Figure 6.1, does not account for the self-overlapping architecture but only how the user will experience. This was then further developed into the design seen in Figure 6.2. This design also shows the puzzles made for the wall as well as some of the models that are used.

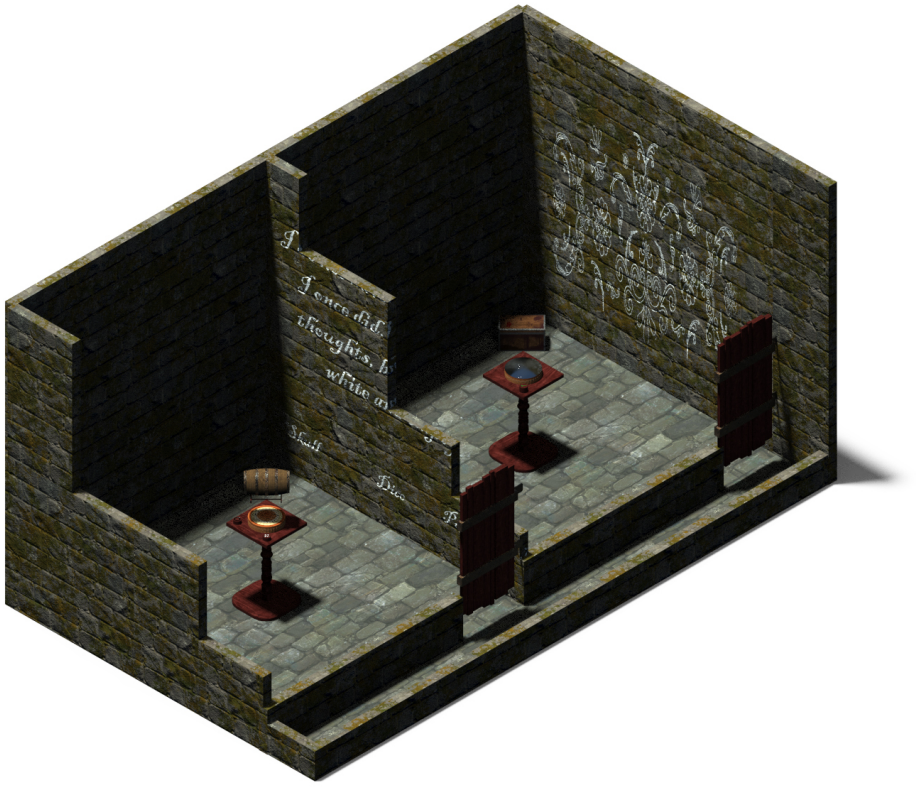


Figure 6.2: An initial design of the room layout

6.1 Audio

In the project we have used several royalty free audioclips from *freesound.org* to add to the immersion of the entire experience[7]. In both rooms we used an ambient sound on a continuous loop.

7. Conclusion

Even though some parts of the design hold up rather well, there is still space for improvements. The part of the project that would benefit most from a revised design is the puzzles. Since no prior tests regarding the puzzles were conducted before the testing for the project was done, the issues with the puzzles the users faced had not come to light before. Look in the project report section 6.2 on thoughts on how the project could be further improved.

Same goes for the implementation. While it does its job well, it could be rewritten to have less dependencies, and be more streamlined.

Bibliography

[1] *Nintendo - Official Site*. URL: https://www.nintendo.com (visited on 11/29/2017).

[2] Nicolaj Evers, Sule Serubugo, and Denisa Skantarova. "Self-Overlapping Maze and Map Design for Asymmetric Collaboration in Room-Scale Virtual Reality for Public Spaces". In: (May 2017), p. 60.

[3] E. A. Suma et al. "Leveraging change blindness for redirection in virtual environments". In: *2011 IEEE Virtual Reality Conference*. Mar. 2011, pp. 159–166. DOI: 10.1109/VR.2011.5759455.

[4] *Choose Hand Safety - CPWR | What is My Hand Size?* URL: https://choosehandsafety.com/choosing-hand-tools/hand-tool-size (visited on 11/29/2017).

[5] Nora E. Scott. "Egyptian Cubit Rods". In: *The Metropolitan Museum of Art Bulletin* 1.1 (1942), pp. 70–75. ISSN: 0026-1521. DOI: 10.2307/3257092. URL: http://www.jstor.org/stable/3257092 (visited on 11/29/2017).

[6] *Average Hand Size - Your resource for Hand Size, Palm Size and Finger Length Averages*. URL: http://theaveragebody.com/average_hand_size.php (visited on 11/29/2017).

[7] *Freesound*. URL: https://freesound.org/ (visited on 12/11/2017).

A. Appendix A

In this appendix are the survey from the hand awareness test and the script used to conduct the test. The results from the survey can also be found here as an .csv file.

A.1 Script

Hello, we are group 17531 and we want you to participate in 4 small tests.

All of the tests consists of you grabbing an object from table, walking to the other side of the table and the placing down the object again. The object will be colour coded, and there will be a colour coded circle on the table. It is important that you try to the best of your ability to place the object in the middle of the circle with the corresponding colours pointing in the right direction. You only get one try in each test, so as soon as you have placed the object, the test is complete.

To start the test please go to the middle of the room.

Virtual (visible) For this test you will be using the Vive controllers to pick up a virtual object. So pick up the object and walk to the other side of the table by going left.

Virtual (invisible) For this test you will use the Vive controllers to pick up a virtual object. You will not be able to see the controllers or your hands. So pick up the object and walk to the other side of the table by going left.

Physical (visible) In this test you will be wearing a tracker on each hand, and be picking up a physical object, which is also represented in the game. You will use your hand to pick up the object. So pick up the object and walk to the other side of the table by going left.

Physical (invisible) In this test you will pick up a physical object, that is tracked in virtual reality. Even though you are wearing trackers, you will not be able to see your hands. You will use your hand to pick up the object. So pick up the object and walk to the other side of the table by going left.

A.2 Post-test survey

Hand awareness post-test survey

Virtual object with visible controller

How well were you able to use your hands to interact with objects in VR?

	1	2	3	4	5	6	7	8	9	10	
Very poorly	○	○	○	○	○	○	○	○	○	○	Very well

Do you have any comments on this type of interaction?

Your answer

Virtual object with invisible controller

How well were you able to use your hands to interact with objects in VR?

	1	2	3	4	5	6	7	8	9	10	
Very poorly	○	○	○	○	○	○	○	○	○	○	Very well

Do you have any comments on this type of interaction?

Your answer

Physical object with visible controller

How well were you able to use your hands to interact with physical objects represented in VR?

	1	2	3	4	5	6	7	8	9	10	
Very poorly	○	○	○	○	○	○	○	○	○	○	Very well

Do you have any comments on this type of interaction?

Your answer

Physical object with invisible controller

How well were you able to use your hands to interact with physical objects represented in VR?

	1	2	3	4	5	6	7	8	9	10	
Very poorly	○	○	○	○	○	○	○	○	○	○	Very well

Do you have any comments on this type of interaction?

Your answer

Share your thoughts

Which method do you prefer to interact with objects in VR?

- ○ Virtual object - visible controller
- ○ Virtual object - invisible controller
- ○ Physical object - visible controller
- ○ Physical object - invisible controller

Please elaborate on your answer above

Your answer

SUBMIT