Creating 3D environments of paintings used for learning geometric concepts

Daniel van der Born, Yan Li Jiang
Student IDs: 4007441, 3941701
Utrecht University

Game and Media Technology
Master Thesis

July 3, 2017

Supervisors:
First supervisor: drs. V.H. Jonker.
Second examiner: prof. dr. R.C Veltkamp.

Abstract

Recreating single 2D images to 3D environments is one of the classic computer vision problems. Specifically recreating paintings introduces the problem of having an incorrect perspective, which makes reconstruction with automatic and semi-automatic methods impossible. This means that the paintings have to be reconstructed manually using 3D modeling software. We present an application that contains a 3D environment to teach children about perspective. In order to teach perspective, tasks are created that are derived from assignments in the CITO exams. The tasks that children have to do, derived from the assignments, can be used in any reconstructed painting. One painting has been recreated and duplicated to show that task generation is random and works in any painting. The application shows there might be educational potential for learning about both perspective and the painting itself. However, the group of children that took part in the experiment is too small to make concrete statements about the results.

Keywords: 3D Reconstruction, Perspective, Paintings, Gamification

1 Introduction

Using virtual environments to teach children about different kinds of concepts, ranging from language to math and everything in between, has been done for many years. These kinds of environments have shown that they help with teaching the concepts that they are supposed to teach [15]. Two big challenges arise in creating such an environment:

- What kind of interactions do you create that teach the concept
- How do you keep the children engaged, so they don't get bored with the application

From the applications available there have been environments that can be used to teach about perspective, but that is not the main focus of these applications [2]. An application that is tailored to teaching perspective to children has not been created yet.

In this thesis we describe the application that we created to teach children aged 9 to 11 about perspective. The aim for the research is to create a gamified 3D environment of 3D reconstructed paintings for the children to use. This environment then uses the interactions we created to learn about perspective. What kind of interactions we created and why is also described. Further, we describe the environment we used to create the application and why we chose this environment. During development we followed the build research methodology, as seen in section 1.1. Every framework in the application is kept at a generic level, to ensure it can be used in any reconstructed painting model that adheres to the implemented tagging system.

The paintings to reconstruct are from the collection present in the Boijmans museum, our application is made in collaboration with the museum and the Freudenthal Institute.

1.1 Research Methodology

Computer science consists of many methodologies for research. From these methodologies we follow the build methodology as described by Elio et al. [5].

The aim of this methodology is to build an application that demonstrates that it is possible to build said application with the features we want to investigate. The application being build must be either completely new or have features that have not been shown before in other applications.

1.1.1 Design

The software that will be created is designed first. A modular design is ideal, because it makes it easier to introduce new concepts. It also helps with simplifying testing. Small interfaces that work with each other on a modular level should be created, so that each interface can be reused if needed.

1.1.2 Reusing components

Not everything has to be made from scratch, many components are already available and ready to use, like: physics, rendering, collisions and many more. Using these components saves time. When making the decisions on which components to use, the license that the component has been distributed under has to be taken into consideration.

1.1.3 Programming language

Choosing the programming language that is appropriate for the application that is going to be built is important, since a language made specifically for a certain task will decrease development time. Choosing a language should be decided by looking at required run-time speed, expressiveness, reliability and the available libraries.

1.1.4 Testing

Testing is an essential part for building applications. By using sets of input and output pairs to test if the code is working, future changes can be tested. Ideally this testing system is automatic.

1.1.5 Documentation and version control

In order for people that did not create the application to understand the code at a glance, having documentation is essential. It is also essential to have a version control system. This allows people to easily access the code and documentation from anywhere with an internet connection.

1.1.6 Evaluation

The created software has to be evaluated, which is done by comparing it to existing software. The different implemented functionalities are compared against functionality in similar software. This evaluation is used to verify if the claims made about the system are correct.

1.2 Research questions

The main question we want to answer is:

- How can a 3D environment be created from paintings, which can be used by children to learn about geometric concepts?

In order to answer this question, a few subquestions have to be answered:

- What kind of interactions can be used to teach perspective?
- How can the different aspects in the environment, like gameplay mechanics and objects, be generalized so it can be applied to any 3D environment?
- How can gamification be applied on the 3D painting environment for teaching about geometric concepts?
- What kind of methods can be used to create the environment?

There are multiple objectives to fulfill that will help with answering the questions. The application needs to have a good user experience. Creating this user experience is done by testing different versions of the application with the target group. They provide feedback as to what works and what does not, which is used to improve the user experience.

It is also important for the application to run smoothly on the target platforms. Difficult platforms to optimize 3D environments for are WebGL and smartphones. An application can be optimized by reducing the amount of draw batches and keeping the amount of triangles that are being rendered as low as possible. There has to be an adequate mix between the amount of triangles and the shape of the object as seen in the original painting. This is a performance versus quality optimization problem.

To learn the geometric concepts, the user has to be able to do some sort of interactions ranging from pulling and pushing objects to tasks for the user to do that teaches perspective.

Ultimately the application has to be easily accessible, which WebGL and smartphone versions will ensure. It has to be able to run on school computers, this includes Chromebooks.

If at any point a new painting has to be reconstructed, then adding this newly reconstructed painting to the application should be as simple as possible. Some kind of automatic system should be made to handle this.

1.3 Structure

In section 2 previous relevant work is stated. Section 4 shows how the application is made. The main experiment and a pilot study are explained in section 5 and the results of the main experiment are shown in section 6. The results section also explains the method used to evaluate the results, in accordance to the research methodology. A discussion in section 7 answers the research questions, shows the future work and has a general reflection on the thesis. Lastly, the conclusion is presented in section 8.

2 Related work

An overview is given in this section on the different aspects that are related to our topic. First, the different techniques of 3D reconstruction are shown, then the previous work done on recreating paintings in 3D is presented, followed by the geometric concepts relevant to perspective and at the end of the section the previous work on gamification is explained.

2.1 Reconstruction methods

For the reconstruction method we are specifically looking for a way to minimize user interaction while still being able to reconstruct everything. It is necessary for a method to allow for the reconstruction of human figures and other complex forms and it needs to be able to deal with incorrect perspective. If a method has those two criteria, then it can be used with the painting reconstruction in the thesis.

2.1.1 Automatic 3D reconstruction

The techniques for automatic 3D reconstruction allow for the recreation of 2D images to 3D environments without user input. This type of reconstruction can either be done with multiple images or from a single image. Recreating an image to 3D when using a single image is a well known computer vision problem.

In order to reconstruct depth in a single image Hassner et al. [7] use a database of objects that are from one classification, like hands or heads. Their reconstruction algorithm uses an example-based synthesis approach. It uses the patches present in the database from which the depths are known, to estimate the depth of an image.

A similar approach by Zhang et al. [17] uses a trained dataset of RGB-D images. The category the object on an image fall under is estimated and reconstructed based on the RGB-D images in the same category. The 3D structure is estimated from the 2D appearance by looking at the variations in rotation, the 3D structure itself and texture of the object.

Using a Markov Random Field (MRF) Saxena et al. [13] infer patches in a 2D image. These patches have a 3D location and rotation. The MRF they created is a trained network and it models both the depth and the relationships between the different parts in the image.

Vouzounaras et al. [16] created an algorithm that uses line segments from either the orthogonal planes present in an indoor picture or the line segments from the exterior of the building to derive the 3D geometry of the single image. The perspective distortion present in images is removed in order to create a more accurate 3D model.

If multiple images are available from an object, then a 3D model can be created based on the different viewpoints. Aubry et al. created a method where 2D images of a location are aligned, no matter the age, lighting and drawing error differences in the images. This allows for larger 3D models to be reconstructed. Dealing with the complexity of larger 3D models is done by representing the model with a smaller set of distinctive visual elements. Types of objects can then be detected. The elements present in images have features that are weighted. These learned visual elements then match the 2D image with the 3D model. The images can be photographs, drawings, paintings, etc, important is that there are multiple pictures available from different perspectives.

These automatic techniques need images with a correct perspective in order for the techniques to work correctly. However, older painting have an incorrect perspective. This makes automatic techniques unusable for painting reconstruction.

2.1.2 Semi-automatic 3D reconstruction

Semi-automatic reconstruction techniques allow for easier object reconstruction. The amount of user interaction involved in these techniques are mostly limited to drawing lines around objects, which are used to determine the 3D geometry.

Models can be reconstructed from drawings when the drawing is orthographic or perspective. Lee et al.[11] created an algorithm that reconstructs the model in the drawing to 3D automatically. Their method requires a unit cube made by the user in order for the reconstruction to work.

Zou et al. [18] improved on the method where an object in an image is reconstructed based on lines that were traced by the user. The lines do not have to be 100% accurate, the lines can

be sketched with a certain amount of alignment deviation. When the lines are used to reconstruct the sketched object to 3D, the alignment errors are fixed automatically.

Semi-automatic techniques also require the image to be in a correct perspective in order to recreate it into a more acurate 3D model.

2.1.3 Manual 3D reconstruction

Manual reconstruction of images requires the user to do everything. Recreation is done in 3D modeling software, which have a learning curve before it can be used easily by the user. Recreating an image manually requires a significant amount of time, but allows for perspective correct and complete recreations.

The "Arnolfini Portrait" was recreated by Jansen et al. [10]. This painting has an incorrect linear perspective. This incorrect linear perspective was corrected by manually correcting the orthogonals. The orthogonals can be corrected by creating a single vanishing point. A horizon is superimposed where the vanishing point is, this horizon is used to measure the "Arnolfini Portrait" and recreate it with a correct perspective.

Carrozzino et al. [2] also used manual reconstruction to reconstruct a painting that has an incorrect perspective. In manual recreation the missing parts are filled by the user, which introduces a human factor. These spots are normally filled based on the surroundings. The reconstruction was created as an educational tool, which could be used to teach about perspective.

	Automatic	**Semi-automatic**	**Manual**
Needs correct perspective	Yes	Yes[1]	No
Amount of user input	None	Drawing edge lines	Everything
Human factor	No	No	Yes
Limited by object type	Yes	Yes[2]	No

Table 1: Different reconstruction methods and the criteria they fulfill

2.2 Geometry

The geometric concept relevant for our topic is perspective. Perspective encompasses the realm of the location of objects, how objects relate to other objects in terms of location, space and angle.

The CITO exam, which elementary school students have to take, has a specific section with questions that test the knowledge on perspective. These questions range from determining from which direction a photo was taken to how a photo will look like when standing at a certain spot.

2.3 Gamification

Applications that are made to teach can benefit from using game design elements to keep the user engaged. When game elements are used outside of the gaming context it is called gamification. Sombrio et al. [15] created a comprehensive overview on the subject of gamification. In gamification the theory of flow is an important concept. This theory of flow consists of eight components: the activity that requires a certain skill, the merging of action and awareness, some clear goals, feedback, concentration on the activity, sense of control, loss of self-consciousness and an altered sense of time.

With the feedback from users Christel et al. [3] improved on the design of their gamified application. Faghihi et al. [6] used quizzes to show that their gamified application can teach concepts to the users.

[1] If the perspective is incorrect, the object will not be reconstructed properly.
[2] It can not reconstruct human figures for example.

3 Application Components

The application for this project is designed first. During this designing, decisions are made on: how the paintings are recreated, what engine to use, what platforms to export to and what kind of interactions could be made to teach children about perspective. In this section the results of these decisions and why we made them are explained.

3.1 Painting Reconstruction

Previous attempts at recreating paintings into 3D have all resorted to manual reconstruction, because of a wrong perspective in the paintings. We have the same incorrect perspective problem and have decided to manually reconstruct the painting.

For the reconstruction we used the 3D modeling software Blender, which is free. We had previous experience with Blender as well. The software was used to recreate the painting "Gezicht op de Mariaplaats en de Mariakerk te Utrecht" from Pieter Saenredam, seen in figure 1. The reconstruction we created is a simple version of the painting.

Figure 1: Painting to reconstruct

3.2 Game engine

In order to develop a game, a development environment that helps with reusing components is needed. For games in specific there are three possible environments. There are many game engines which contain everything needed to build gaming applications. Game frameworks, which are similar to game engines, have less reusable components and are in pure code.

For fast prototyping, a game engine is the ideal choice. Game engines have varying learning curves. Once the learning curve is over, the development of the game application costs less time than with another game engine or framework.

When deciding on a game engine, it is important to look at the used programming language, the experiences of other developers with the engine, the learning curve and the amount of help that can be found online in terms of tutorials and questions. The type of game engine we need for our application has to adhere to two criteria: it can export to multiple platforms and it is free to use or it has a certain limit before you have to pay. Ideally the engine uses a programming language we are familiar with. In our cause it also needs to be able to render 3D graphics.

Cowan et al. [4] made an overview on the various frameworks and game engines currently in use by developers. The survey found that the most popular game engines are Unity and Unreal. Deciding on the game engine then depends on the features present in the engine, which are seen in table 2.

It was a tie between Unity and Unreal, but for our application we decided to implement it in the Unity game engine. Unity is used more often by serious game creators [4] and has a larger community which can be asked for help. We also have previous experience with the engine, allowing for faster development.

	Unity	Unreal	MonoGame	Source Engine	GameMaker
Physics	Yes	Yes	No	Yes	Yes
3D rendering	Yes	Yes	Yes	Yes	No
Networking	Yes	Yes	No	Yes	Yes
Shaders	Yes	Yes	Yes	Yes	Yes
Shadows	Yes	Yes	Yes	Yes	No
Wide export options	Yes	Yes	Yes	No	Yes
Familiar language	Yes (C#)	Yes (C++)	Yes (C#)	No (VScript)	Yes (GML)
Free version	Yes[3]	Yes[4]	Yes	No	Yes[5]

Table 2: Different game engines and frameworks and their features

Unity allows us to export the game to multiple formats easily and it uses the C# language for programming. It also allows us to easily implement components like physics, collisions, camera controls, user interface (UI), visual effects and more.

3.3 WebGL

The application has to be easily accessible for elementary school students, such students usually have access to the internet at school. This means that WebGL, a 3D OpenGL renderer for the web that requires no plugins, is the ideal platform to use for the application.

There are a few limitations for WebGL, namely the amount of CPU power that a computer needs to run the program on the web. Unity converts the application to WebGL, which for the GPU side means near native speeds, because the GPU is directly used for rendering. However, the CPU runs JavaScript code compiled by asm.js. The performance of the CPU side depends heavily on the used browser. There is also no multi-threading in WebGL yet [1].

With the recent introduction of WebAssembly the difference between standalone applications and web applications is getting smaller, but WebAssembly is not yet supported by any mainstream browser [12].

The computers used in elementary schools are not built for 3D gaming applications, this in combination with WebGL means that optimizing the application is essential.

3.4 The interactions

In order to teach the children about perspective, We create four types of tasks. Two of the tasks are based on assignments present in previous CITO exams, while the other two tasks are made to acquaint the user with the controls in the application.

A tutorial task teaches the user how to move, look around, sprint, open the menu and enter a painting. Once the user has entered a painting, the task is completed. By completing this task the user will have used at least the movement buttons to move around, the mouse to look around and the mouse buttons to enter the painting.

The movement task is used to acquaint the user with the controls used to move objects around in the painting. In this task the user has to move a building from one location to a different marked location. The object to drag and the location to where it has to be dragged are shown in the dialogue and in the game world with indicators when the task is active. Once the object is placed onto the new location, the task is completed. If the user completes the task then the dragging controls and the movement controls inside the painting have been used by the user.

After the movement task both tasks for learning perspective are unlocked. The first of these tasks is called the placement task. In this task the user has to move the buildings that are in the wrong locations back to their correct locations as seen in the real painting. The task is loosely based on assignments in the CITO exam where the children have to determine from which perspective a photo is taken of a given scene or how the scene looks like when a photo is taken from a certain point. [9] [14]. In the application itself we decided to change it to moving objects back to their original place. This requires the user to look at the real painting and determine where an object

[3]When yearly income exceeds 100000 USD, upgrade to Unity Plus for up to 200000 USD yearly income, else the Unity Pro version has to be used.

[4]When more than 3000 USD is earned, 5% of the gross product revenue is payed to Epic Games.

[5]The free version is limited and games created with it are not allowed to be commercially sold.

should be in 3D, which they also have to do when determining how the scene will look when seen from a certain point. The other task involves moving a person to a position where a building can not be seen by the person. This task is based on the CITO assignments where an image is given of multiple people standing at different distances from a scene. The assignment is to determine how each person sees the scene. In the application, it is changed to moving a person to a location where it can not see a certain object. This way the user will have to move to locations where the object can not be seen and place the person object there.

4 Implementation

Creating the application involved the creation of multiple frameworks working in tandem with each other. The frameworks are kept at a generic level to ensure it can be used with any reconstructed painting. In this section we present what frameworks we made, how we made them and how they work.

4.1 Controls

Unity has a built-in input system, that exists in two varieties. The InputManager uses names for the input and these names are linked to buttons on either a keyboard, a controller or both. The name of the input and the buttons it belongs to are all determined in the InputManager section of Unity. This system also allows for axis of values, which means in case of analogue sticks that the values will range from -1 to 1 with all possible values in between. It is also possible to get the absolute values of -1 and 1 without the values in between, this depends on the used functions to get the stick input. If keys are used for such axis movements, then the number goes from 0 to 1 at a constant speed. It can also move from 0 to -1 if the negative button is pressed. Which button is negative and which one is positive is determined in the InputManager. A new project in Unity has a few premade buttons in the InputManager, two of these are the horizontal and vertical axes as seen in 2. The other system gets specifically one key, button or screen touch which are always either false for not pressed or true for either one frame or the entire duration it is being held depending on the used function. In this section, the different controls present in the application are explained, both what the controls do and how they were implemented. These controls interact with the different frameworks in the application.

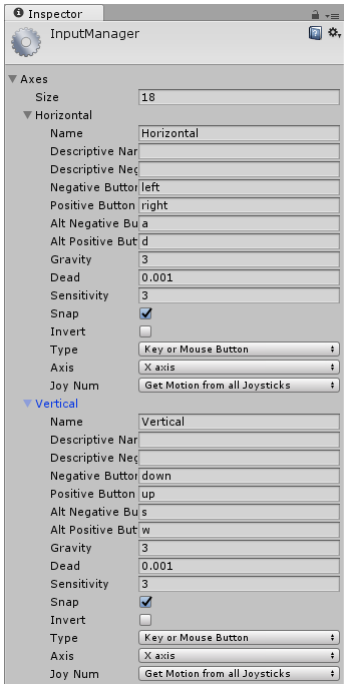


Figure 2: Unity InputManager

4.1.1 Movement

The user can move in the application by moving the mouse to look around and using either the WASD keys or the arrow keys to move around as seen in most applications with a first person camera. In the application there are two types of movement: one for inside the museum and one for in the painting.

For the museum controlled character the Character Controller component that is built-in in Unity is used. The Character Controller can move an object with two functions: Move and SimpleMove. We used the SimpleMove function, which does not consider the y-velocity, but it does use the x- and z-velocities. Gravity is automatically applied when using the SimpleMove. Using the input from the horizontal and vertical axes, we get the x- and z-velocities ranging from -1 to 1. These ranges are multiplied by a custom speed variable. Using these velocities, where y-velocity is zero, the character is moved. The SimpleMove does not move the character abruptly, instead the speed of the character linearly interpolates toward the value it is given.

The movement controls inside the painting does not use the axis input. Instead, the linear interpolation towards the maximum speed is done manually with the built-in Lerp function. The position is manually changed as well. Each game object transform in Unity has a position variable that that can be changed and the transform also has multiple vectors corresponding to directions. These vectors, forward, left, right, backward, all point in the direction local to the object. This means that if a cube is rotated by 90 degrees on the x-axis, then the forward vector points downward. The direction vectors are used to make the character in the painting move forwards, no matter the rotation of the camera. The camera its forward vector is multiplied by the speed *and* by the delta time. Delta time is used to make the movement frame rate independent. The right vector of the camera transform is used to move sideways.

Once the position of the painting character is updated, a check is performed to keep the character object inside the 3D painting its boundary. This is done by using the the position of the painting and the size of the painting. For example, the character object can not go past the wall at the left side of the painting, because when its x position becomes smaller than the position $painting\_x - \frac{painting\_width}{2}$, the character is put at the side of the wall. The $painting_x$ is the x position in the middle of the painting. Because of using these checks instead of the built in collision system for this specific part, the character object can not tunnel through the walls with fast movement. The checks for the other sides are similar, bringing the total checks to six checks per frame. It is also possible to use Unity's built-in collision system. However, when used the player moves too fast and tunnels through the walls. The collisions between the painting character and other painting objects is done via the built-in physics and there are painting object that the user can tunnel through when moving too fast. Too fast in this case is the speed of the sprint mode. It is not possible to tunnel through all the objects, the tunneling problem is limited to the thin painting objects, like the simple rectangle shaped objects that have nothing extra.

When the left shift button is pressed, a higher maximum speed is set for the character. This acts as a "sprint", allowing the player to traverse the environment significantly faster. We double the maximum speed in this sprint mode.

Looking around by using the mouse is the same for both the painting and museum characters. The InputManager in Unity has "Mouse X" and "Mouse Y" axis that can be read. These inputs correspond with the amount of mouse movement in either the x- or y-axis, ranging from -1 to 1. A mouse sensitivity variable controls the speed the camera rotates at, it has a default value of 5. The horizontal rotation gotten from Mouse X is used to rotate the object itself on the y-axis, meaning it rotates to the left and right. Rotating an object in Unity is done with the object its transform. A Unity transform has a function called Rotate. Giving this Rotate function the Mouse X multiplied by the mouse sensitivity it then rotates the object and because the camera is a child object of the character object, the camera rotates as well. Vertical rotation is kept in a variable that does not reset every frame. At every frame the amount of mouse movement in Mouse Y multiplied by the mouse sensitivity is subtracted by the value present in the vertical rotation variable. The value is then clamped, which makes it so the player can not rotate 360 degrees vertically. The value in the vertical rotation is then given to the local rotation of the camera instead of the object. The transform also has a local rotation variable, which is the rotation the object has in the space it is in. Because of the hierarchical structure of objects in Unity, an object has a local rotation and a world rotation. Local rotation is the rotation in reference to its parent, if there is no parent then the local and world rotation are the same. When not using a function such as Rotate, the local rotation itself can be set using degrees by using the built-in quaternions. A quaternion can be

created from degrees by using the Euler function present in Unity quaternions.

4.1.2 Dragging

During the creation of the application, two dragging systems were made. The change was made based on feedback and personal preference. In both systems an object is moved by moving the mouse around. The difference is in how the object reacts to being dragged by the mouse. To start dragging an object, the user has to hold the left mouse button and look at a moveable object. Only one object can be dragged at a time. The controls framework casts a ray every frame from the middle of the screen and this ray is used to determine what type of object is hit, some of the controls the user can perform depends on the object that is hit. When a ray hits a moveable object, then the dragging controls can be used.

In the old dragging system, the object can be moved around the camera. Moving the object to the left and right is done based on circles. The angle of the camera on the y-axis is known. This angle is then used to calculate the next angle and thus the next position on the circle, where the center is the camera and the distance to the object is the radius. The angle changes based on horizontal mouse movement. An object can also be pushed and pulled. This was done using the vertical mouse movement. However, pushing the object meant pushing the mouse forward which in turn moved the camera vertical rotation upwards as well. The speed at which pushing and pulling happens depends on the distance of the object to the camera. The further away the object is, the faster it is moved. This causes a disorienting effect making the dragging confusing. The movement itself is done with the Translate function present in Unity transforms. These move the object based on the given vector.

The new dragging system was made to make dragging feel more natural. When an object is being dragged, the distance is calculated between the camera and the object. A ray is cast through the middle of the game screen. Unity has a function for cameras called ScreenPointToRay, which we use with the point at the middle of the screen. From this infinitely long ray, we grab the point on the ray located at the distance between the camera and the object. The object that is being dragged is then linearly interpolated to this point on the ray. Because the position on the ray changes when the camera is moved by mouse movement, the building "follows" the middle of the screen while being separated by the previously calculated distance from the camera. Moving the object like this is similar to moving the object on the surface of a sphere, where the center of the sphere is the camera and the radius is the distance between the camera and the object. While dragging, the distance between the object and the camera can be changed. When the user uses the scroll wheel the distance either increases or decreases depending on the scrolling direction. In our implementation the distance becomes shorter when the scrolled towards the user and larger when scrolled away from the user. This scrolling has replaced the pushing and pulling that was previously done with mouse movement. The distance between the camera and the object can not become smaller than a set minimum distance. In the application this minimum distance is 4, which corresponds to 4 unity blocks. A unity block is the size of a standard unity cube that can be created as a primitive type in the engine. Just like with the painting character, an object that is being dragged is also checked to make sure it stays in the bounds of the painting. Both versions of the dragging system are shown in figure 3.

The user can also use the right mouse button and drag. In this case, the object is rotated on the y-axis. This rotation is also done with the built-in Rotate function and the amount of mouse movement on the x-axis. The camera rotation that normally happens is disabled when an object is being rotated and re-enabled once the right mouse button is let go.

4.1.3 Resetting

Pressing the R button while inside the painting or when looking at the painting representation, created by the framework in section 4.3, causes trapdoors to open underneath each moveable object in the painting and it places each object back to its starting position and rotation. which loads in the 3D painting models and places them into the scene. It is also possible to reset an individual object, by pressing R while looking at the object. This type of single building reset is handled in its own framework explained in this section and it does not involve the use of the trapdoors.

The trapdoors are automatically created when the framework for the 3D painting placement turns objects with a @D tag in the name into moveable objects. When a moveable object is created, its bounds is used to scale the trapdoor prefab. A prefab is a premade Unity game object

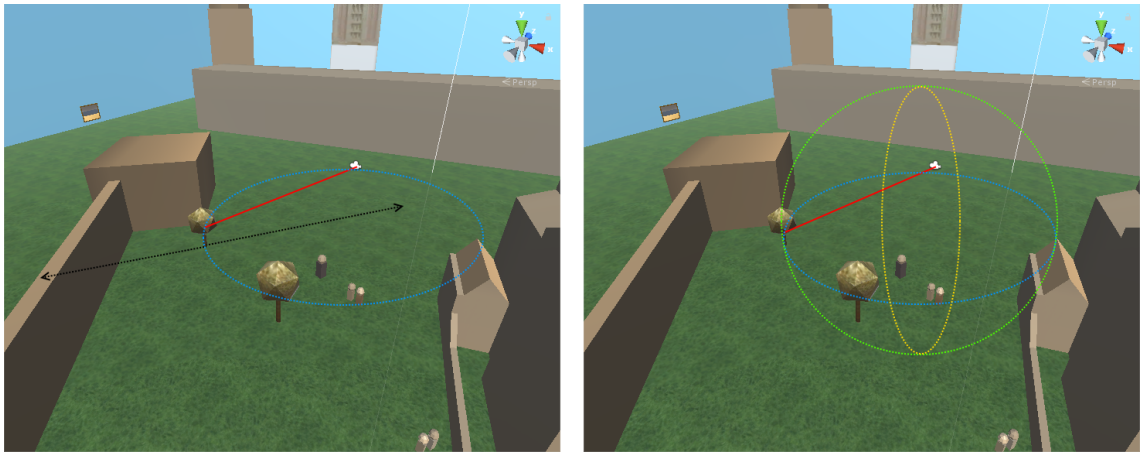


Figure 3: **Left:** old dragging controls. The object pointed at by the red line can be moved in a circle around the camera and it can move forward and backward based on the mouse movement. **Right:** new dragging controls. Red line is a ray cast. Object can be moved around the sphere surface where the red line is the diameter.

structure, as seen in figure 4. When scaling, 125% of the width and depth of the moveable object size are used. For the height 100% of the object height is used. The standard scale of the prefab is 1 for every dimension, the 1 is the size of 1 Unity block and because all the scaling is based on Unity block putting the scale of the trapdoor equal to the size of the object means that the trapdoor has the exact same size. Giving the width and depth the extra 25% ensures that the trapdoors are clearly visible when pressing the reset button, this can be seen in figure 5.

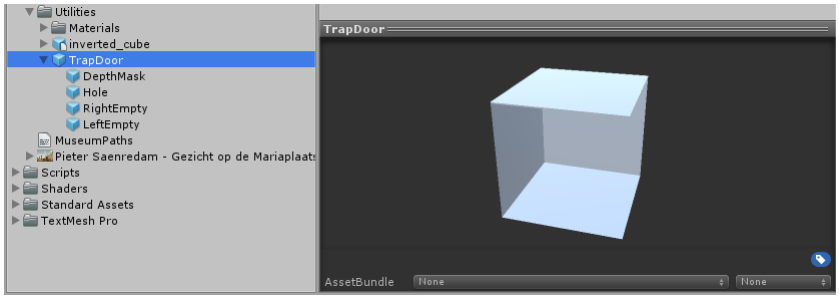


Figure 4: Trapdoor prefab

Making the objects fall is done by disabling the bounds check for the floor for each moveable object. There is no check for the ceiling, unless the object is being dragged. These checks are done on a per building basis, which means that when the building has reached an y-position of $approximately(current\_y - \frac{object\_height}{2}, painting\_y)$, where $approximately$ is a built-in Unity function that returns true when two floating point values are close to each other, then the floor bounds check is turned back on for that object.

The trapdoor prefab has two objects called "left" and "right". These are the doors of the trapdoor and when R is pressed the doors are opened. This is done by linearly interpolating the rotation of the local z-axis rotation of the two objects to a target angle. For example: the target angle for the left door is 0 degrees, because at 0 degrees the trap door is "open". The right trapdoor also goes towards 0, but it goes from -90 to 0 instead of 90 to 0.

Once the top of an object has gone past the top of the trapdoor, the trapdoors are closed and the object is put above the painting. Checking if the top of the object has gone past the other top is done by checking the y-position of both. The top of the object is known by taking the y position of the object that is located at the center and adding half of the height of the object to this position. This top position is compared to the y position of the painting and not the actual top position of the trap door. This is done because the tops of the trapdoors are located slightly above the y-position of the painting to make sure there is no z-fighting between the floor and the top of the trapdoor. Before the the object is put above the painting, the trapdoors are first closed.

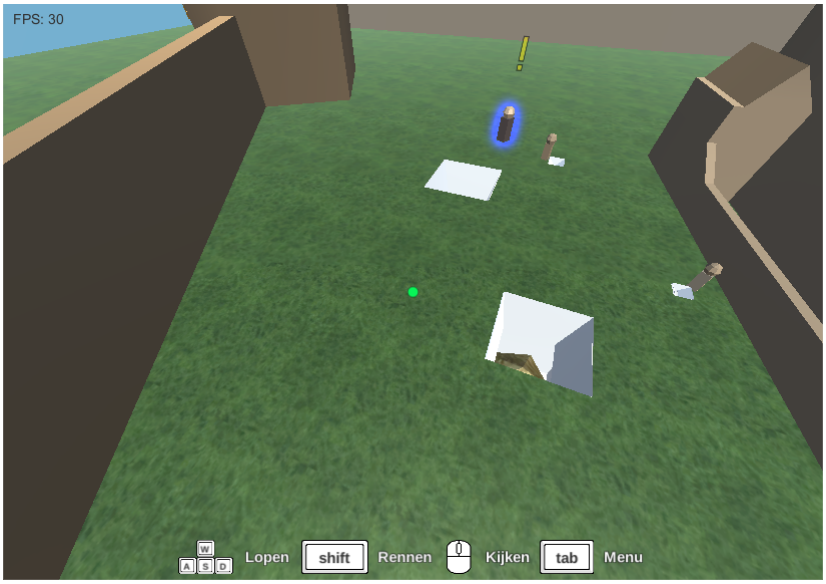


Figure 5: Painting reset with trapdoors

This is done by doing the exact opposite of opening the trap doors. The target angle becomes 90 degrees instead of 0 degrees for the Quaternion lerp. Just as before, for the right trapdoor the target angle becomes -90. It has to be -90 degrees, because it opens and closes the opposite way down in comparison to the other trapdoor.

After the trapdoor is closed, it is deactivated. Deactivated game objects in Unity are no longer rendered and updated. The object itself is then put above the painting, but not directly above, instead it is put two times the height of the painting above said painting. The x and z positions are set to the x and z the application started with for that object and the same is done for the rotation. A force is added to the moveable objects, which is equal to the height of the painting times the downward vector to make sure it moves downwards at a fast pace. Forces can be exerted on rigid-bodies using the built-in AddForce function present in the rigid-body component. This force together with the gravity makes the object fall down onto their original spots and because of the bounds checking that happens it can not tunnel through the floor. The bounds check for the ceiling is disabled during this reset period for the moveable object, else the objects would spontaneously be right underneath the ceiling.

While the moveable objects are falling, they can not be dragged by the user. Once an object has reached the ground, it can be dragged again. Once all the objects are done falling, the painting can be reset again. The outlines explained in section 4.2.4 are disabled when an object is being reset and they are turned back on once the falling is done.

As seen in figure 5, you can not see the painting floor through the trapdoor as if there actually is a hole present in the floor quad. However, this is not the case and instead there are some depth changes that cause this effect. In order to create such an effect in Unity, you need a depth mask object the size of the hole object. This depth mask has a custom shader that sets the ColorMask value, which is a built-in Unity shader value, to zero. A zero value means that nothing is rendered, but the object itself is still present and active in the scene, it can still interact with other objects. The actual pass of the shader is empty. Lastly, the Unity shader tag system is used, this basic shader gets a "Queue" tag that is set to "Geometry-1". This tag section can have different kinds of tags, but the Queue tag is a predefined tag that controls when an object is rendered. The lower the number given to the Queue tag, the earlier it is rendered. Standard opaque objects have a standard Queue value of "Geometry" which is equal to 2500. The value that is rendered earlier than Geometry is the "Background" value, which is equal to 1000. Since the depth mask has Geometry-1, which has the value of 2499, it is rendered before all the other opaque objects. The actual hole object and objects in the painting have the standard Unity shader, but with a Queue tag value of Geometry-2. As such, the objects and the hole are rendered first, then the depth mask is rendered on top of that and lastly the painting walls, ceiling and floor are rendered on top of everything. This gives the illusion that there is an actual hole in the floor when the trapdoors are opened. If the depth mask is not present, then you do still see the floor going through the hole.

As mentioned before, resetting an individual object does not use the trapdoors. Instead the position and rotation are linearly interpolated to the starting position and rotation. This happens in its own function separated from the painting framework.

4.1.4 Changing painting view

When looking at the 2D representation of the 3D painting, which is created in the museum environment explained in section 4.3, the user can change the perspective of the painting. When the controls framework detects you are hitting an object with the "Painting3D" tag, it can change the perspective when the user presses a number ranging from 1 to 5. The perspective transition happens in the framework that deals with 3D painting placement, from section 4.2. Unity also has a built-in tag system. Each object can get a tag and the developer can use this to determine what type of object has been hit with a ray or is hit in collisions and objects can be searched for based on the tag they have. The tags are created at the same location where the layers are created.

When the perspective changes, an empty game object in the middle of the painting rotates by x amount of degrees. The angle it gets depends on the pressed number. This rotation happens by linearly interpolating the current quaternion to the target quaternion. Since the camera that looks into the 3D painting, giving the 2D representation, is a child of the empty game object, it rotates as well. In Unity when the parent object rotates, the children are rotated around the parent object by the same amount of degrees the parent object is rotated. This means the camera rotates in a circle around the empty game object. The top down perspective is handled slightly differently. Instead of rotating on the y-axis, the camera rotates on the x-axis to 90 degrees. This causes the camera to look downwards. The y-rotation is set to 0, meaning it always sees the top-down view the same way. Finally the projection of the painting camera is turned into an orthographic projection, which is done in order to make it true top-down. When projection is set t perspective, you can not see the entire painting in the top-down view. When in orthographic projection mode, the orthographic size of the camera is set to $painting\_camera\_y - painting\_y$, with this size the 2D representation can show the entire painting in the top-down view. Each of the numbers correspond to a different perspective. The available perspectives in order for the keys 1 to 5 are: front view, left side view, back view, right side view and top-down view. The different perspectives can be seen in 6.

Alternatively, when the user is inside the painting pressing one of the numbered keys the user is instantly transported and rotated to have the correct position for the perspective associated with the number. Changing a perspective is not limited to outside the painting. When inside a painting and exiting the painting with a different exit point than the one the user entered, the painting perspective is automatically changed to show the perspective the user left the painting at. Meaning that when a user enters the painting in the front view and exits at the back view, then the painting in the museum will show the back view.

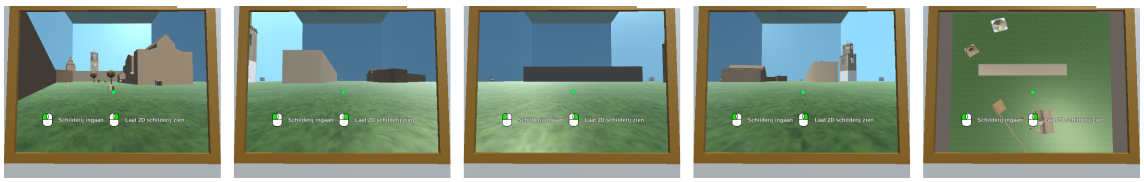


Figure 6: The different perspectives. From left to right: front, right, back, left, top-down

The user can also switch between looking at the 2D representation of the 3D painting or the real 2D painting the reconstruction is based on. This is done by pressing the right mouse button while looking at the painting representation. This is handled in the controls framework. In Unity, game objects can be set to active and inactive using the SetActive function present in all objects with MonoBehaviour scripts.

4.1.5 Menu

When pressing the tab key, the user opens the menu. The menu is explained in section 4.6.2 The different options in the menus use the built-in Unity UI system for input. A button script in Unity allow for functions to run when the user clicks on the object containing the button script. For the

Figure 7: Switching between the real painting and the 2D representation

menu this means deactivating the current menu and activating the menu the tab in the menu links to with the button script. These tabs can be seen in figure 18.

4.1.6 Dialogue

Looking at person objects enables the user to use the spacebar. When pressing the spacebar, the dialogue system is activated and shows the text present in the person object. This dialogue system is explained in section 4.5. While in a dialogue the user can also select different dialogue options, which are done the same as with the menu where the button functionality is created using the built-in UI system. Pressing spacebar while in normal dialogue allows the user to skip text and advance to the next part of dialogue.

4.2 3D painting models

The 3D painting placement framework has multiple responsibilities concerning the 3D painting models.The framework loads the 3D models present in the sub folders of the Paintings folder as Unity game objects. The prefab for the trapdoor is also loaded here. Using these game objects, the framework creates C# representations of the painting. In this C# class the 3D model is parsed for information. The 3D model has a reference to the texture of the painting it is based on, which is used by other frameworks.

4.2.1 Entry points

Five entry points are defined, these are the points the user will end on when entering the painting. The points correspond to the five different perspectives and they are located at the lower center of the walls and the middle of the ceiling. These points are calculated based on the width, depth and height of the painting and the position of the painting. The painting position is determined by a given offset. The offset is calculated as $(i+1)*200$, where i is the index number of the current 3D painting that is being added to the Unity scene. This way paintings are separated from each other by 200 Unity blocks and this allows for multiple paintings to be rendered at the same time without paintings overlapping with each other. The offset is used on the y-axis of the starting position, the x and z positions are both 0. To then get the entry points, 5 similar but different looking vectors are created, each follows the same principle. For example, the entry point from the front view is calculated as: $x = 0, y = painting\_y + 0.6, -depth/2 + \frac{1}{2}$.

4.2.2 Parsing Child Objects

The different child objects present in the 3D model are parsed and given a tag. When parsing the 3D model, the object with the largest width and depth is found. This is the floor object, which is kept separate from the other objects to ensure it is not seen as an object inside the painting. For every child object, the tag at the end of the name denoted with the @ symbol is parsed. If there is no tag at all, then the object gets the "Static" tag and is added to the static objects list. If an object contains the P tag, then the object is added to the person objects list. When a T is encountered, the object is either added to the list of taskable static objects or the list of taskable dynamic objects, this depends on if the D tag is present as well. In case there is a D tag, the object gets the "Dynamic" tag and is added to the dynamic object list. The other frameworks use

these four different list. At the end of the parsing the object with the largest depth and width is removed from the static objects list. This does assume the floor does not have a @D tag.

The dynamic and static objects are represented in C# with their own classes. They inherit from an abstract generic object class. The dynamic objects get rigid-body components, the trapdoor from section 4.1.3 and they get the outlines explained in section 4.2.4. The generic object has a reference to the object itself and it knows if it can be used in tasks. The static objects do not have any extra information.

The different tags have different meanings and are used for different purposes. An object that has the @D tag is an object that the user can move and rotate. If an object has the @P tag, then the dialogue framework will recognize it as a person object that can give tasks to the user, as seen in section 4.5. When a @T is present, the object can be used by the task framework as objects that will be used in the tasks that are generated, which is explained in section 4.4. A static object, meaning there is no @D present, is a decoration in the painting that uses Unity's built-in physics system for collisions with objects with the rigid-body component. Combination of tags can also be present, a @DT tag is a moveable object that can also be used in tasks.

Person objects are not related to the generic object, instead it is its own framework. A person object can have a task to give to the player and it has a blue glowing outline.

4.2.3 Painting Exterior

Every 3D painting generates its own walls and ceiling. These objects of the painting exterior are created, scaled and rotated. In Unity, primitive objects can be made in the code by using the CreatePrimitive function. The walls and ceiling are quads. The creation of the walls is done with a for loop: first is determined if we are dealing with an even number, then the object is created. The width of the exterior object gets the width of the painting if the number is even, else its width becomes the depth of the painting. Wall height is calculated as two times the height of the 3D model, which allows for movement above the painting. Once scaled, the object is placed at a depth of either $painting\_z + painting\_depth/2$, where the $painting_z$ is the z position of the center of the painting, or it is placed at $painting\_z + painting\_width$ depending on if the current wall index is an even number or not. The x position is set to $painting\_x$ and the y position is set to $painting\_y + wall\_height/2$, where the division by 2 is necessary, because the position point of the wall is at the center of the quad. The last step rotates the wall piece around the center of the painting by $i * 90$ degrees, which places the pieces at the edges of the painting. Here i is the index of the wall piece. There are four walls, meaning the walls are rotated by 0, 90, 180 or 270 degrees. The system is illustrated in figure 8 and it shows how the first two wall pieces are placed in painting. The ceiling is created similarly, but its x-axis rotation is set to -90 degrees to make the quad look downward. The y position of the ceiling is $painting\_y + wall\_height$.

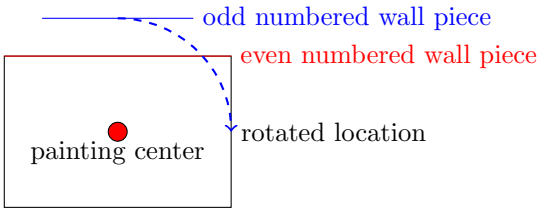


Figure 8: Wall placement. **Red line:** wall piece where $i = 0$. **Blue line:** wall piece where $i = 1$

4.2.4 Object Glowing Outlines

Some objects in the painting get a glowing outline. Person objects get a blue outline, dynamic objects get an orange-yellow outline and when looking at objects with an outline, they turn green, as seen in figure 9.

These outlines are a visual effect created with shaders. First, every object that can have an outline uses the standard Unity shader, but with an added tag in the Tags section. In the Tags section the custom tag "Glowable" is set to "True". A different custom shader also gets the glowable tag set to true. This custom shader returns the given color.

A second camera renders the scene again, but only renders the objects with the glowable shader tag set to true. However, instead of rendering these objects with their standard shader,

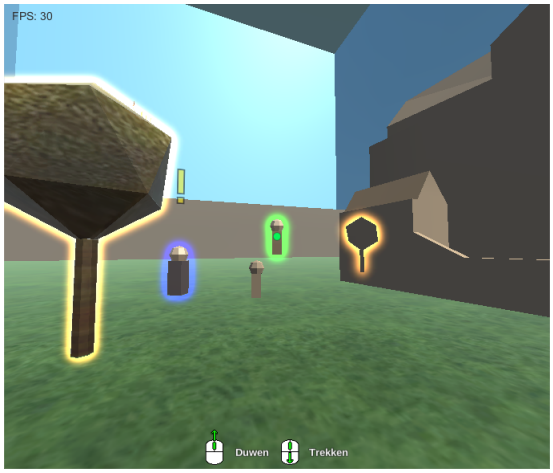


Figure 9: The different outlines

we want to render them as solid colors with the custom shader that has the same glowable tag. This can be done in Unity with a feature called shader replacement. This second camera renders the glowable objects with the shader that only returns the color, which is done by using the SetReplacementShader function present in Unity cameras. This function searches for the tag it is given, "Glowable" in this case, and checks if the value of the tag in the current object shader corresponds to the value in the tag present in the shader it will replace. If the tags and values match, then the object is rendered in this second camera with the new shader instead of the shader the object has normally. This second camera renders the background as a black color, which allows outlines to be invisible when they are set to black as well. The rendered image of the second camera is given to another custom shader, which blurs the given image with a Gaussian blur. The blurred image is also downscaled to half the size of the non-blurred image, which is done for better performance. Unity has the OnRenderImage function for monobehaviour scripts that gives access to the rendered camera image. In this function the Blit function can be called which applies a material on the rendered image and returns the new image. A copy of both the non-blurred and blurred images are kept, which are used in the last step where the blurred image is subtracted from the non-blurred image. This subtraction happens in a final custom shader. The final custom shader has references to the blurred and non-blurred images created previously. The resulting image from this shader is an outline and this shader is applied to the main camera with another Blit call, giving the glowable objects a glowing outline.

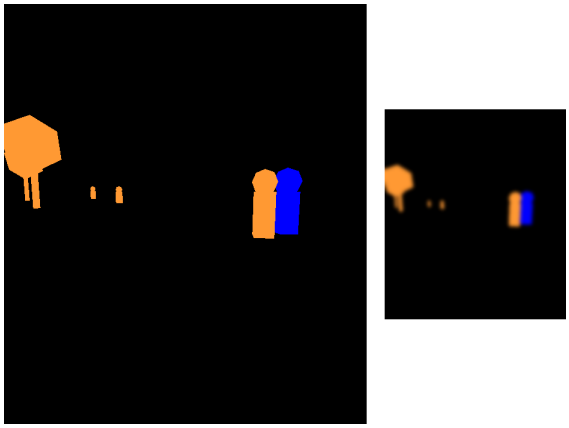


Figure 10: **Left:** the second camera rendered image. **Right:** blurred version which is used in the subtraction

4.3 The Museum

The user does not start in a reconstructed painting, instead the application starts in a museum environment where the paintings are hanging on the walls.

This museum environment is procedurally generated, based on a tile system. Currently the generation is limited to a large hallway, which can be seen in figure 11. However, it can be extended to include various rooms and corners as seen in figure 12. The tile grid of the museum has multiple tiles with each knowing their 2D position, their type and they know which painting they have and if they do not have a painting, then this painting index is set to -1.

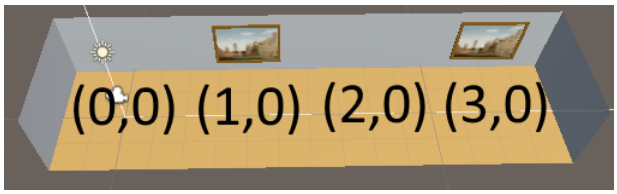


Figure 11: Generated museum with tile grid positions visible

There are six types of tiles in total: a floor tile, an empty tile and four tiles where paintings hang either north, west, east or south of the tile depending on the tile type. The current museum generation loops through the size of the 2D tile grid, which is set to the amount of 3D paintings multiplied by 2 for the number of columns and 1 for the number of rows.

When looping through the 2D grid, each time the current tile index is an even number it is a floor tile and if the index is an odd number the tile type where the painting is on the north side of the tile is picked. After determining what kind of tile type we have, the tile type is parsed into Unity objects. This consists of multiple parts.

First, the paintings hanging on the walls are generated. The position of the painting is calculated as:

$$x = tile\_x * tile\_size, y = y\_offset, z = (tile\_y + \frac{1}{2}) * tile\_size - \frac{painting\_width}{4}$$

Where the y offset is a predetermined height, currently set to 1.25 unity blocks above the museum floor. The y-coordinate of the tile is used as a z coordinate for unity, since the environment is 3D and the z-axis is the depth axis. The painting width is predetermined as well, the current value is 0.1 Unity blocks. It corresponds to the dimension towards the wall and by subtracting it from the z coordinate located at the center of the north side of the tile, it is placed right against the wall.

During this for loop, each time a tile type is found that has a painting, it gets the painting index for the reconstructed painting that will be linked to the tile. This painting index is then used to create the 2D representation that hangs in the museum. First, the quad is created with the built-in Unity CreatePrimitive function and it gets a material with an unlit shader, which is built-in in Unity. A camera is created that is put in the reconstructed painting placed at the middle of the south painting wall. This corresponds to the front view of the painting. The camera targets its rendering to the texture in the newly created material. This can be done in Unity by setting the targetTexture variable present in Unity cameras. Another quad is created the same way as the first quad, but its texture is set to the painting texture of the real painting, which is saved in the reconstructed painting.

The quads get a frame existing of four scaled cube primitives. This is done in the exact same way as the walls system from section 4.2.3. The only difference here is the scaling. Its x scale is set to either: $x\_scale = painting\_quad\_width$ or $x\_scale = painting\_quad\_height + edge\_width * 2$ depending on if the current frame edge index is an even number or not, and its y scale is set to $edge\_width$. The rotation happens around the center of the painting quad, but this time the rotation happens around the local z-axis instead of the y-axis. Once the frame is done, the entirety of the frame and the painting quad are rotated based on the direction of the tile type. When it's the tile where a painting hangs on the south wall for example, it is rotated by 180 degrees on the y-axis.

2D representations of the museum hall present in the reconstructed paintings are generated after the generation of the 2D reconstructed painting representation. These museum representations are created in the same way as the painting representations, but the positioning is based on the painting wall placement system.

Once all painting representations are generated, the museum floor and ceiling are created. They are both quads scaled with $x\_scale = grid\_width * tile\_size$ and $y\_scale = grid\_height * tile\_size$ The floor is rotated by 90 degrees on the x-axis so it faces upwards and the ceiling by -90 degrees so it faces downwards.

Last step is generating the walls. The double for loop through the museum grid is done once more, but this time the loop starts at $x = -1$ and $y = -1$ instead of 0. In this loop three tiles in the grid are grabbed: the current tile, the tile to the right of the current tile and the tile above the current tile. When a tile is grabbed that is outside the grid size, an empty tile is returned. There are two major tile situations: either the current tile is empty or its not. If it is empty, and the tile to the right or above the current tile are not empty, then walls have to be placed on the edges of the either the right or upper tile. When the upper tile gets a wall, the position is :

$$x = tile\_x * tile\_size, y = \frac{museum\_height}{2}, z = tile\_y * tile\_size - \frac{tile\_size}{2}$$

This corresponds to the position on the upper lower edge of the upper tile. The wall is rotated on the y-axis with a given rotation, which is determined in for double for loop. When the upper tile is not empty, then the rotation is 180 degrees, so the quad looks away from the empty tile. In case of the right tile the wall position is similar, but the calculation after the multiplication symbol are swapped between the x and z coordinates. Rotation for the right tile when the current is empty, is -90 degrees. When the current tile is not empty, but either the right or upper tile are, then the same calculations happen with different rotations. 90 degrees for the right tile and the upper tile wall stays at 0 degrees.

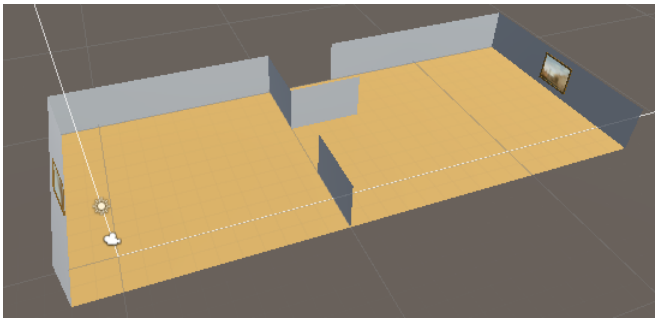


Figure 12: More complex museum example, using a predefined tile grid

4.4 Task System

A generic task framework is created to handle different types of tasks. This framework consists of multiple constructs. The different interactions described in section 3.4 are turned into C# classes using the task framework. The task framework makes use of the dialogue framework explained in section 4.5.

4.4.1 Task XML files

Each interaction is converted into an XML file that describes the task and contains the dialogue structure for the task. The dialogue part of the XML file is explained in section 4.5.1. Besides the dialogue, a task XML file contains the task title, short description, detailed description which contains a longer explanation about the objective and it contains a difficulty number ranging between 1 to 5. The root element of a task XML contains an attribute that shows the task type. Creating a new task starts with the XML file, where these tags must be present. A snippet of an XML task file can be seen below, the full XML file for this task can be found in appendix A.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<task type="movement">
    <title>Verplaats het object</title>
    <description>Zet het object op een andere plek.</description>
    <detailed>Verplaats het object en zet het op de goede plek.</
   detailed>
```

```
    <difficulty>1</difficulty>
```

Listing 1: Task XML

4.4.2 The Generic Task

All tasks in the application have to inherit from the abstract generic task class. The information present in this generic task is used by other frameworks.

The generic task parses a given XML file and saves the information present in the file into multiple variables. The dialogue part of the XML file is also saved in a Dialogue C# class, explained in section 4.5.1. Parsing XML can be done in C# with the built-in System.Xml namespace.

Different task states are tracked by the generic task. A task can be the set to active, it can be accepted and it can be completed. For parsing it keeps track if the given XML file was a valid task file, where valid means it has all the tags mentioned in section 4.4.1 which have to be non-empty and the difficulty tag needs to be a number.

When a task is completed, accepted or activated, a callback function is called, which can be used by other frameworks to call their functions when one of those three states are changed in a task. C# gives the developer the possibility of callback functions with delegates. Simple to use delegates are the Action and Func delegates, which can contain any function that adheres to the Action or Func generic parameters.

There are a few functions that have to be overridden in the classes inheriting from the generic task. Each child class has its own conditions for when the task is completed, which if true sets the task being completed to true. If the parsing went alright, then there is another validity check present in each child class and each task has different conditions showing the reconstructed painting the task will be put in is valid for the task or not. In order to move the camera during dialogue, each child class has to override a function that calculates the position and rotation the camera will have when the dialogue arrives at an attribute containing "camera-view", as seen in section 4.4.4.

4.4.3 Tutorial Task

The very first task users will come across is the tutorial task. Dialogue for this task starts when the application has been loaded and it can not be declined by the user. When this task is created, the validity check always returns true. There are no special conditions for the task to be valid. The task can be completed by entering any painting.

Entering a painting automatically sets a boolean for if we are inside a painting to true. The index of the current active painting is always known and can be accessed by anything, but can only be set to a different value by the controls system, because when the user looks at the painting representation the system knows with the casted ray that a painting is hit and using the position of the painting we know on what tile the painting is and thus what the painting index is.

Once completed, the task starts the final dialogue. This dialogue shows the name of the painting and the author. The movement task is also generated when the tutorial task is completed.

4.4.4 Movement Task

In order for this task to be generated in a painting, there has to be at least one object with the tag @DT and one static object. If this is present, then the task is valid and is generated. The movement task can be seen in figure 13.

The task randomly selects one of the @DT objects present in the painting it is created for and it also randomly selects one of the static objects. Once these objects have been selected, a goal is generated. The goal itself is a quad scaled to the size of 3 Unity blocks. Placing the goal is based on the position of the randomly selected static object. From this static object one of four offsets is randomly picked. This direction is a number from 0 to 3 and just like the painting wall placement system, the goal is placed at the position of the static object and then pushed away with either half the depth of the static object if the direction is an even number or with half the width if its not even. The goal is then rotated around the static object with the Unity RotateAround function, where the degrees to be rotated on the y-axis is determined by $direction * 90$. An object indicator is created, which targets where the @DT object is located and both an object marker and indicator are created with them both targeting the goal object. How these object markers and indicators work is explained in section 4.6.6.

Completing this task requires the user to move the random @DT object on top of a goal object. When a rectangle spawning the goal overlaps with a rectangle spawning the width and depth of the moving object, then the moving object is on top of the goal. The x and z values of the objects based on position, width and depth are compared. With these values, the left most x value, gotten by subtracting half of the width of the object from the current x position located at the center of the object is checked and it has to be smaller than the right most x value of the goal, which is calculated the same but half the goal width is added to the x position of the goal instead of subtracted. These two values are the minimum x value of the moving object and maximum x value of the goal object respectively. Then the right most x value of the moving object has to be higher than the left most x value of the goal. The highest and lowest points on the z-axis are also compared similarly, bringing the total to four checks. If all checks return true, then the moving object is overlapping with the goal and the task is completed. When the task is active and the moveable object is being dragged, then the marker and indicator for the goal are set to active and when the moveable is not being dragged the marker and indicator are deactivated. The indicator for the moveable object is always active when the task is active.

In this task, when its dialogue comes across the "camera-view" attribute in the dialogue and it has a value of "target", then the camera moves towards the position calculated with a custom function. This function calculates the position diagonally above the given object. The camera is also rotated with the built-in Unity function called "LookRotation", which is present in Unity quaternions. This function is given the direction towards the object from the position the camera will be at after the linear interpolation has finished. Both the position and rotation are linearly interpolated. The same happens when an attribute value of "goal" is found, which then moves and rotates the camera towards the goal object. If any other value is found in the attribute, then the camera does not move.

Once the task is completed all the specific task objects that were generated are turned off and two new tasks are generated for the painting: the placement and position tasks.

Figure 13: Movement task, with object indicator

4.4.5 Placement Task

Creating this task requires the painting to have at least one moveable, or @D, object.

Before the generation process starts, arrays are created for: multiple object indicators, original start positions for objects, colors of the outlines and for the moveable objects themselves. These arrays have the size of 50% of the number of moveable objects in the painting.

The generation process itself is a loop through 50% of the moveable objects. First, random x- and z-coordinates are calculated. These random positions are limited to the floor area of the painting: $random\_x = (r * painting\_width) - \frac{painting\_width}{2}$ where r is any random value between 0 and 1, including the 0 and 1, the same is done with the z but then using $painting\_depth$. A random moveable object is chosen from the painting, and a list of object index values ensures that a random object that has already been used in the loop can not be selected twice. The gotten random number excludes values that have already been selected before. The object indicator is created that targets the random moveable object. A rectangle is created with the Rect struct, which is built-in in Unity. The x- and y-coordinates of the rectangle are the upper left corner of the rectangle. The center of the created rectangle has the random x- and z-coordinates as its center x- and y-coordinates. Half of the object width is subtracted from the random x-coordinate

to get the left most x and half of the object depth is subtracted from the random z-coordinate to get the highest z value. The 2D position is then the upper left corner of the rectangle.

This rectangle with the size of the area of the object is then intersected with each rectangle created from the static objects. The Unity Rect struct has an Overlaps function that returns true if the two rectangles intersect. If the object its rectangle intersects with one of the static object rectangles, then the randomly chosen position would cause the object to be put on top of another object. However, we do not want objects to be placed on top of other objects during the random placement, thus we mark it as invalid and it redoes the loop and the random object index value that was previously calculated is not added to the list of already used indices. If none of the static objects intersect, then the moveable objects are checked in exactly the same way, with the addition of not checking the moveable object when it checks against itself. When the checking is done and it is still a valid position, then the current starting position of the object is saved. The starting position is then changed into the random position and its current position is also set to the random position. This ensure that, when the user presses the reset button from section 4.1.3, the object does not automatically fall onto the correct position. The object itself is saved at the current index position of the objects array created previously and the index is added to a list of all the object indices that have been used up till that point.

Figure 14: Objects with red markers need to be placed on the correct locations

In order for the user to complete the task, all the objects have to be placed back to their old starting position. Instead of placing the object on the precise location, two rectangles are created where one is the rectangle for the moveable object and the other rectangle has the correct starting position as its center and has a predetermined width and height. Currently the width and height is set to four Unity blocks. This means the object will be placed correctly once the rectangle is within two Unity blocks of distance of the correct position. If an object does not intersect with the rectangle around the correct position, then its outline and indicator colors are changed to red. Once it intersects, the outline color is returned to orange and the marker is returned to green. If the user misplaces the object again, then the colors are back to red. These red outlines can be seen in figure 14. Once all objects intersect with their correct position rectangles, the task is completed. In oder to know where the objects have to go, the users can see the real painting in the detailed description located in the task menu from section 4.6.2. As a reward for completing the task, the user gets the current painting placed into the gallery section of the menu explained in the same section as the task menu explanation.

4.4.6 Position Task

For this task, seen in figure 15, to be valid there needs to be at least one static object and one person object that can be moved. If such objects are present in the painting, then the static object has to be visible for the person object else it is still invalid. To check for visibility, each moveable person object in the painting is checked against every static object until a combination is found where a person sees the static object. If the person already has a task, then the next person is grabbed. When going through the static objects, random objects are chosen with the

same exclusion if it had been chosen previously as seen in the generation method of the Placement Task. The person object and static object are then used in the function ObjectIsObscured, which is a custom function that returns if the first object can not see the second object. This function first calls another function CalculateRayCastHitPositions, which is also a function we made. Five y-coordinates are calculated that are positioned from the top of the static object to the bottom. Then the y-coordinates are used to calculate the x-coordinates based on an interval. The amount of x-coordinates an y-coordinate can be paired with is predetermined. Currently each y-coordinate can have five x-coordinates bringing the total to 25 coordinates. The interval is then calculates as:

$$interval\_vector = y_0 - \frac{static\_object\_width}{2} * right\_vector$$

Where 0 is the first of the five y-coordinates, *right_vector* is a 3D vector pointing to the right $right\_vector = (1, 0, 0)$. Finally, the interval becomes:

$$interval = interval\_vector - \frac{interval\_vector\_length}{n}$$

Where n is the number of x-coordinates per y-coordinate and *interval_vector_length* is the length or magnitude of *interval_vector*, which is a built-in property that every vector type has in Unity. With this interval, a loop is done through all the y-coordinates and for each y-coordinate the x-coordinate is calculated. The resulting position is saved. The total position is calculated as:

$$position = y_n - \frac{static\_object\_width}{2} * right\_vector + (interval * m, 0, 0)$$

The first part before the + sign is the same as the interval vector, with the difference being the used y-coordinate. The n is the current y-coordinate in the loop and the m is the index of the current x-coordinate being calculated. A loop goes through the calculated positions and a line is cast from the person object towards every calculated position using the built-in Unity physics Linecast function. If any of the cast hits the randomly chosen static object, it means the object is visible to the person object. The big difference between this task and the other tasks is that the task giver is a part of the task itself, which requires this validation method to make sure it does not get created by accident.

Figure 15: The person must not see the tower

In order to complete the task, the ObjectIsObscured function has to return true. Which is done by moving the person object to a location where all casts do not hit the static object.

For generation, this task creates a task indicator that targets the static object that has to be hidden from the person object's view.

When it comes across the camera-view attribute in dialogue, it uses the same functions as the placement task, but only when the value "target" is found. This moves the camera to the static object.

4.4.7 Managing And Generating Tasks

A MonoBehaviour class handles every available task and intertwines with the task generation. First, it creates a list of tasks for every painting and then the different task XML files are parsed. A task generator is created, which is a class with only one available function and that function creates a task from the given XML file for the given painting. In C# you have access to an "Activator", which has a function called CreateInstance. We use this function in combination with C# generics. The CreateInstance returns an object that is of type T, where T is of the class Task and because of inheritance this includes all classes inheriting from Task. If the task was not valid, a null is returned by the task generation function.

The tutorial task is added to a separate list that contains the tasks that can be accepted in the museum environment. A call back is added that checks if the tutorial task is finished and if this happens and if there are no tasks currently in the painting. This means that after the tutorial task if the user enters a new painting, it will have tasks to do.

Each time a task is generated, the user interface for this task is created as explained in section 4.6.2 and in section 4.6.5, callbacks are created for the task which ensure the task user interface is functional when the task is active. The interface also takes care of setting the active state of any available task.

4.5 Dialogue

Accepting or declining tasks, getting information about the task and general discourse happens with the dialogue system. There are two parts in the dialogue system: the dialogue that occurs when the user is talking to a person object and the dialogue that appears when the user is near a person as a text bubble above the person's head.

4.5.1 Dialogue XML File

The content of the dialogue is present in XML files and the task XML file from section 4.4.1 uses the dialogue XML format to give the actual task to the user and to give a detailed explanation on the objective of the task.

This dialogue format consists of multiple XML tags. First, each piece of dialogue is written in the dialogue tags and each dialogue tag has an action attribute. The value of this attribute is used by the dialogue system to continue a dialogue conversation. Inside the dialogue there are multiple variations on the dialogue that can happen with this action and inside the variations there are one or multiple line tags with the dialogue content inside them. There can also be an option tag at the end of the variation, which has one or more decisions inside it that is used by the system to create a branching dialogue. Every decision also has an action attribute, which corresponds to the next piece of dialogue that will start once the option is chosen. There is also an optional "click" attribute in decisions, which can refer to a function that has to happen. Currently the only possible value is "accept" which accepts the task.

```
<dialogue action="start">
        <variation>
            <line>Oh, hallo.</line>
            <line>...</line>
            <line>Hé, zou je me ergens mee willen helpen?</line>
            <line>Ik probeer iets te verplaatsen, alleen dat lukt
   mij niet. Zou jij dat voor mij willen doen?</line>
            <option>
                <decision action="yes" click="accept">Ja,
   natuurlijk.</decision>
                <decision action="no">Nee, niet nu.</decision>
            </option>
        </variation>
    </dialogue>
```

Listing 2: Dialogue snippet

There is also a floatingDialogue tag, which have the same action attribute values as the dialogue tags. As an example, when the dialogue system is at the "start" action, then when the user is

near the person object it will show the text in the floatingDialogue tag in the text bubble above the person object's head. Inside a floatingDialogue tag there can be multiple variations and each variation has the text the text bubble will show.

```
    <floatingDialogue action="start">
        <variation>Hmm, hoe zou het er uit zien als dat daar zou
   staan...</variation>
    </floatingDialogue>

        <floatingDialogue action="other finished">
                <variation>Bedankt voor het helpen!</variation>
        </floatingDialogue>
```

Listing 3: Floating dialogue snippet

The entire task XML the snippets are from can be found in appendix A.1.

4.5.2 Managing Dialogue

A MonoBehaviour class manages all the dialogue in the application and its first purpose is to create the dialogue interface, which are based on a multitude of prefabs. For the text bubbles, each person object gets a copy of the text bubble prefab. The text bubble itself is a piece of UI with world coordinates instead of screen space coordinates. When creating the main dialogue, there are two that are used by everything dialogue related: the UI that shows the text and the UI that shows the different options a user can choose as seen in figure 16. For each painting a loop is performed that goes through all tasks in that painting. This loop first checks if there are still person objects in the painting that do not have a task assigned to them and if so a random available task is selected. If this random task already has a task giver, which can happen with the position task, then the dialogue callbacks are created for that task's Dialogue class reference. The task is then assigned to the task giver. If no such task giver is present, then a random person that does not have a task is selected and the same is done as before.

Figure 16: The dialogue system: normal dialogue, decisions and floating dialogue

The Dialogue class has all the callbacks in order to work with the dialogue system and interface. This class parses the dialogue portion of XML files and sets the first action value to "start", which means that every XML dialogue needs to have at least one with the action attribute containing a value of "start". For easy access the floating dialogue is also contained in this class as its own C# class.

The Dialogue class has a function called ProgressDialogue, which continues to the next line in the dialogue. First, its checked if the current line number exceeds the amount of lines in the current dialogue section and if so the dialogue stops. When the dialogue is done a callback is ran. Lastly, the action is set to "other X", where X is the current action name. If there are still lines left, then any "img" tag present in the line is parsed to show an inline sprite. This is done in Unity with the TextMesh Pro asset, which is a free asset that will become built-in. Another callback is ran, for anything happening with the text. If the camera-view attribute is present, then a callback dealing with this attribute is ran. When the line that comes next is an option, then the options are enabled with another callback.

4.6 User Interface

The UI is separated into multiple frameworks, which uses the other frameworks to show the information present in the application.

4.6.1 Generic UI

Most of the screen space UI in the application uses classes that inherit from a generic UI class. This class enables the given UI objects to use a transition effect when enabled. Currently there are two possible transition effects: fading and sliding horizontally. Every piece of UI that uses a class inheriting from the generic UI gets its own container. This is done to enable the transition of the entire piece of UI. For transitions the CanvasGroup component is added to the container. This Unity UI component can set its alpha to 0, causing itself and every child to be invisible. When using this class, it assumes that there are prefabs that have to be instantiated. The path to these prefabs is a params paramater, which is a parameter type that allows for comma separated values of the type associated with the params. In case of UI this is a params string. It is also possible to give an array of strings as the parameter instead of comma separated values. Params is specific to C#, but other programming languages have the same features in different names depending on the language. Each child class also has one or multiple integer values that correspond to the different prefabs it has access to. This is done to make the code more readable. For example, when the controls that are shown have to change based on the object that has been ray cast, then in the code it will say "DRAGGING_CONTROLS" instead of "0".

Which transition effect will be used for the UI is determined in the child class. A function called DetermineAppearanceEffect has to be created in the child classes in which a function is given to the OnAppear callback.

When a piece of UI has the fade effect, it is first determined if it fades in or if it fades out. This depends on the given boolean value. Using a Func that returns a boolean, a callback is created that uses the boolean value given to the fade function to determine when the loop is done. If the fade boolean is true, then the while loop continues as long as alpha is not yet 1, else it loops until the alpha is 0. Inside the loop the alpha is increased or decreased by $active * 0.05$, where active uses the fade boolean to either be 1 if true or -1 if false. Every frame either 0.05 is added to the alpha or removed from the alpha until either 1 or 0 have been reached. This happens in a while loop, but instead of running the entire loop in one frame, one loop cycle is done every frame. This is done with Unity's coroutines. These allow the developer to create loops where the program can return to later instead of having to run the loop until it's done in the current frame. In coroutines, denoted as the IEnumerator in Unity, this is done by using a *yieldreturn* statement. If it says $yieldreturnnull$ then the program returns to that location in the next frame. It is also possible to use built-in functionality as $yieldreturnnewWaitForSeconds$, where the program returns to the coroutine in n amount of seconds. Although these are called coroutines, they all run in one thread.

The slide transition works similarly to the fade transition. Instead of the alpha value, the position of the container is linearly interpolated to the middle of the screen and the object slides from the right side of the screen to the middle. Only the x-coordinate is changed, the y-coordinate stays the same. When the UI is deactivated, the slide transition will move the container towards the left side of the screen from the middle of the screen.

4.6.2 The Menu

Pressing the tab opens up a menu, as explained in section 4.1.5. This menu consists of three tabs where each shows the information of a specific framework. The three menu's accessed by the tabs are seen in figure 18. The menu itself consists of prefabs with button functionality to switch between the different tabs.

A Menu class inherits from the generic UI class. It has three prefabs, each prefab is one of the tabs in the menu. Specific to the menu is a Unity UI slider for the mouse sensitivity. This uses no appearance effect, which means it instantly appears or disappears. The special situation with this specific UI is that when the menu is opened, all other UI elements are set to inactive and when it is closed all the UI elements that were turned off are activated again. The button objects for the tabs automatically get their functionality in a loop. Each UI element goes through its children and if there is a Button component present, it gets the functionality to open the tab that belongs to the current child index.

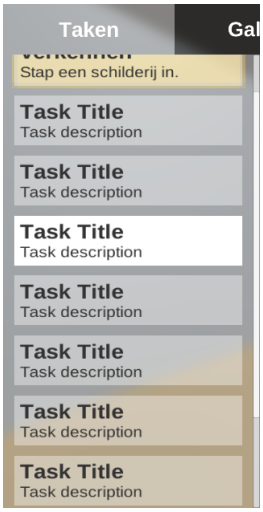
Figure 17: Location UI

Figure 18: In-game menu

In the first tab, the tasks that have been accepted are shown. Adding the task to the menu happens when the dialogue choice has a click attribute in the XML file with the value "accept". The callback that runs when the button is clicked adds the task to the menu, which is done by instantiating the prefab of the task menu content. Each task entry is an instance of a prefab, they are put in a scrolling list which is built-in in Unity UI. These lists automatically work for any platform and can contain UI objects which can be scrolled through. A mask makes sure that only a certain space on the UI screen space shows the entries, overflow from that space is hidden, as seen in figure 19a. This prefab has two text components, which are given the title and the description of the task. It also has a button which runs a function when clicked where the task is set to active and the smaller in-game task UI is activated. This in-game UI can be seen prominently in figure 19b. The text on the right panel is changed into the more detailed description of the task. Clicking on the button again deactivates the task instead. A callback is created for a task completion, when completed the task in the menu turns into a darker gray and it can no longer be set as the active task. It is possible to click on it, which then shows the detailed description in the right panel of the task menu. When a task is clicked that can still be the active task, then an orange outline envelops the task button and the button itself gets an orange color as well. Adding it to the menu is done by setting the parent of the instantiated prefab to the task scrolling list. It is also set to the top of the list, which is done by using the Unity function SetAsFirstSibling. This is done in order to keep all the completed tasks at the bottom of the list and the newest task at the top. After the task has been added to the menu, it is set as the active task and the in-game task UI gets the new task title and description.

In the middle of the menu is a Gallery tab. This section in the menu can show a painting, its title and its author. When the movement task is completed, a prefab is instantiated which is added to a scrolling list similar to addition of tasks to the task menu. It is, however, not possible to click on the gallery entries.

The last menu section shows all the options available to the user. Here the user can change the mouse sensitivity, which is done with a Unity Slider. The mouse speed in the controls system is changed based on the slider. The slider itself has a minimum and maximum value, which are



(a) Scrolling tasks list

(b) In-game task UI

Figure 19: Task Interface

determined in the Unity inspector view. This inspector view can be seen in figure 20. The options menu also shows the different controls the user can use, which are prefabs placed into a scrolling list.

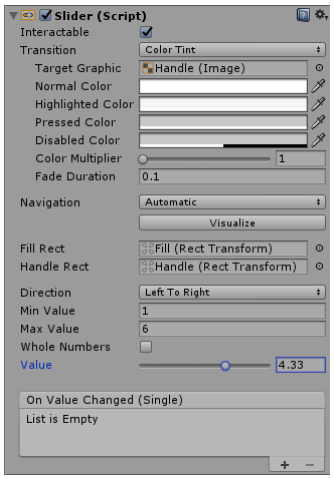


Figure 20: Mouse sensitivity in the inspector

4.6.3 Dialogue Interface

The callbacks in the Dialogue class are used to make the dialogue interface work. This interface is shown in figure 16. When a dialogue starts, which is done with the spacebar as explained in section 4.1.6, the dialogue prefab is activated. The prefab position is determined inside Unity and the Canvas UI object the dialog is contained is set to scale with the screen, which is done with the Canvas Scaler component.

Dialogue text appears letter by letter, which is done with properties present in a TextMeshPro text asset. It has characterCount and maxVisibleCharacters properties. The number of max visible characters is increased by one every 30 milliseconds which makes the letters appear one by one.

When the user can select a dialogue option n amount of options are enabled and the parent game object of the options is scaled to $(300, 75 * n)$, where 300 is the width in pixels and the $75 * n$ gives each option a height of 75 pixels. The position of the options is calculated the same way the goal position from the movement task is calculated, but with an added padding of 16 pixels to leave empty space between the options and the dialogue. Functionality is added to the buttons, where a click sets the action attribute value to the value present in the decision tag and the current

line number the dialog was at is reset to 0, because a new piece of dialogue is started. The accept value for the click attribute is handled by calling the Accept function of the task, which sets the accepted status to true and adds the task to the task menu.

4.6.4 Controls Interface

In the application many different controls are shown, either on the bottom or around the middle of the screen. The UI at the bottom and the UI around the middle are different C# classes that inherit from the generic UI class. The controls at the bottom can show four types of controls, one for the dragging of objects, another for the controls of pushing and pulling, the controls shown at the beginning that stay visible for 10 minutes when there are no other controls shown and the perspective options when looking at the 2D representation of the 3D painting.

As appearance effect the fading effect is chosen. Which controls are shown is determined by the controls system, it is based on which object has been hit with a ray cast and if none are hit and the tutorial controls have not been shown for 10 minutes yet, then the tutorial controls fade in. The content and position of the controls are determined in Unity itself with controls prefabs.

The controls in the middle of the screen also use the fading appearance. These controls can show the controls for entering a painting and switching between the 3D and 2D paintings. Talking to person objects is also shown in the middle of the screen. Which controls are shown in the middle of the screen is also determined by the controls system.

There is one more UI which is controls related and that is the UI that shows the current perspective of the painting. This UI shows up when the user presses a number from 1 to 5, when you start looking at the 2D representation of the 3D painting and when inside the painting one of the numbers 1 to 5 is pressed. The location UI uses the slide appearance effect. When it is activated it slides in, waits for a predetermined amount of time and then slides away. The predetermined amount of time is currently one second.

4.6.5 Task Interface

Besides the task UI in the menu and the task in-game UI that is connected with the menu when you click on the task objects, there is also a task completed UI message. It uses a slide in appearance effect which is similar to the generic UI slide in, but has separate components that slide at different speeds. It waits for half a second before it then slides away again. This task completed UI can be seen in figure 21. When the task completed text slides away, it is also faded away at the same time.

Figure 21: Task completion notification siding in and sliding out

The transparent black background of the task completed prefab moves a quarter second earlier than the task completed text. It happens automatically when any task has been completed, which is a callback created when the task interface for the task is created.

4.6.6 Object Markers

The different tasks lacked a way of knowing which objects to move and for the movement task where it had to move to. In order to let the user know what had to be moved an indicator system is created. There are two types of indicators that can be applied to any Unity game object. These can be seen in figure 13.

An arrow indicator is placed above an object that is targeted by the indicator. There is also an arrow indicator in the screen space UI. There are two situations for this indicator: the user can see the object or the user can not see the object. When the user sees an object, which is determined

with the built-in property isVisible present in the Renderer component of the object, then the indicator above the object is activated and updated. It updates the position of the indicator to ensure it stays above the object even when the object moves. The y-position for the indicator is set to $object\_height + indicator\_height$ and it is scaled with a factor of $distance * 0.1$, where distance is the length of the vector $camera\_position - indicator\_position$. This scaling ensure the arrow is larger the further away the camera is. If the user can not see the object, then the screen space arrow is enabled. This arrow always points towards the object at the edge of the screen. In order to create such an arrow, the position of the object has to be converted to screen space, which is done by using the built-in Unity function present on the camera component called WorldToScreenPoint. The point at the middle of the screen is subtracted from the screen space object position. Using this position, a rotation is calculate using the y- and x-coordinates as tan values. The resulting rotation is subtracted by 90 degrees converted to radians in order to deal with the Unity rotation system. The radians are used to calculates the sin and cos values. These values are used to adjust the screen space object position. The final position of the arrow starts at the center of the screen, multiplied by 0.9, which ensures that there is padding between the edge of the screen and the arrow. If the cos value is positive, then the screen space position of the object is changed to $x = \frac{center\_screen * 0.9}{\frac{cos}{sin}}, y = center\_screen * 0.9$. When cos is a negative value, the same calculation is done, but the $center\_screen$ is negative instead. Finally, the adjusted screen space position of the object is changed to $x = adjusted\_center\_screen, y = \frac{adjusted\_center\_screen}{\frac{cos}{sin}}, z = 0$, where $adjusted\_center\_screen = center\_screen * 0.9$, if the x-coordinate of the adjusted screen space position of the object was larger than the adjusted center screen else the values are negative. Lastly, the center point is added back to the adjusted object screen space position. The indicator position is then set to the adjusted screen space position and the rotation around the z-axis is set to the previously calculated rotation and converted to degrees in order to use Euler function in the Unity Quaternion.

The indicator in world-space uses a standard Unity UI shader, but the ZTest is turned off which causes the arrow to be seen through objects.

A different indicator is the object marker, which marks an object with an Unity LineRenderer component. When this line renderer is enabled, the height is linearly interpolated until the maximum height is reached or the other way around where it linearly interpolates to the minimum height. The maximum height is determined by the class that creates a marker. The movement task is currently the only class that uses a marker and it gives a height of 75% of the 3D painting height. The starting width of the line is set to 3 Unity blocks, but the width at the top of the line is set to 1 Unity block. The color is also given to the marker, which is green for the movement task marker. The color at the top of the line has its alpha set to 0, while at the start of the line it has an alpha of 1. Combined it gives the column from figure 13. The speed of how fast the marker appears is predetermined and is currently set to half a second. Linearly interpolating the top position of the line renderer is done with the built-in SetPosition function, which can set either the starting position or the end position. The end position is denoted with the index 1, while the start position is denoted with index 0.

5 Experiments

5.1 Pilot study

The prototype was used in a pilot to get an initial evaluation of the application's design. For this pilot experiment a group of six children around the age of 9 to 12 use the prototype. The children can walk around freely in a test version of a 3D reconstructed painting. This field test gives a first impression on how children interact with the application. Especially in the early stages of the development process, the feedback and the information we get from observing the children are needed to shape and improve upon the concept.

For this particular setup we look at performance, usability and user experience. The performance is measured by the loading time in seconds and the average frames per seconds in-game. The user experience is evaluated through questioning. The level of usability is obtained by asking for the children's feedback on how the game can be changed or improved.

Each child uses a Chromebook to play the prototype on. The Chromebooks are Samsung Chromebooks from 2012, which contain an Exynos 5000 series CPU with 1.7 Ghz. The Exynos CPU has an integrated graphics card: the ARM Mali-T604, which is also used in smartphones.

Once the web page containing the WebGL version of the prototype is loaded on Google Chrome, the browser on Chromebooks, it takes 38 seconds before the game is done loading. The game starts off having a low frame rate below 30 frames per second, but after 10 seconds the frame rate goes up and stays around 50 frames per second. Different actions lead to different amounts of frame rates. The frame rate can drop below 30 for a few seconds, which happens when the user resets the painting, but it can also stay at 60 frames per second when the player looks at simpler geometry.

The children play for half an hour, while their behavior is observed. When the children are in the painting, they start dragging the objects immediately. They use the dragging controls in a natural manner without requiring our help. Although there is a big prompt showing the pushing and pulling controls when buildings are being dragged, as seen in Figure 22, the children did not use the mouse wheel. The other controls that were visible in the panel in the upper left corner of the screen are also ignored by four of the six children.

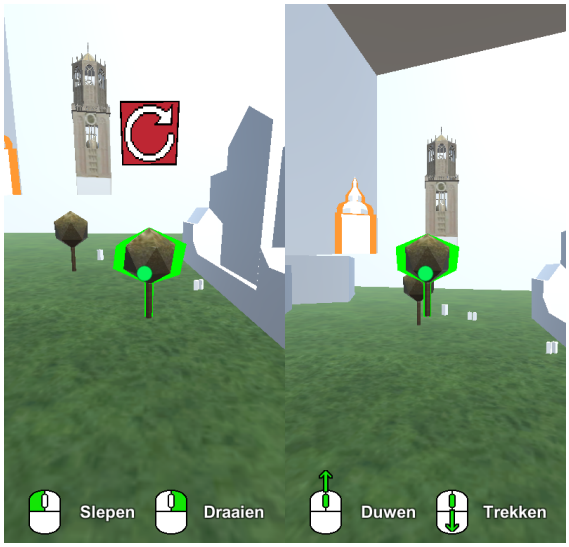


Figure 22: Controls based on current interaction

While playing, the children talk and discuss to each other as to what they are doing. They all experiment with the different possible actions, from moving objects around and placing them on top of each other to resetting the painting by pressing random buttons. The action of rotating is confusing to the children, which was caused by the word that was used to describe the button. To rotate an object it said "rotate", but for the children "turn" is a better term to use and during the test we had to explain that rotating is the same as turning. In the play session the children need to place themselves in such a way that the church is visible and the Dom tower is hidden behind it. The children are able to complete the task quite easily and they understand why the building is not visible in terms of perspective.

At the end of the playing session we ask two questions:

- What can be improved?
- Was it fun to play and why?

All of the children say they enjoyed the prototype and had fun with putting objects on objects. Being able to change the paintings to how they wanted it to look was a big plus to them. Everyone is quite positive and looks forward to how it all looks like in the next version.

They also give feedback on what could be improved:

- More objects to move
- Filling the empty spots
- More paintings to play in
- More types of interaction, like the people waving at you when clicking on them or making them move around the painting

- More details: improving the now blocky parts to more realistic looking versions, e.g. the people in the painting that are now basic rectangular cuboids

5.2 Experiment: Testing version 4.0

The finished application was used in an experiment at an elementary school. 13 children, seven boys and six girls aged 9 to 11, took part in this experiment. Just like the pilot study, the experiment is carried out in a separate classroom.

First the children were explained how the experiment would go. They had to play the application for a maximum of 30 minutes, once the time was up or if the child finished all the tasks present in the application they had to fill in a questionnaire for which they also had 30 minutes. They were not specifically told what kind of tasks they had to do in the application, since we wanted to test our design to see if the implemented task and dialogue system worked.

During the 30 minutes of play the children and the performance of the application are observed. Once again moving, walking around, going into a painting all went smoothly without requiring any questions. Once inside the painting some children asked how that nothing worked. This happened because the children were in a dialogue, but did not know that the space button they had seen in the earlier dialogue would also work here. When the difficult task we had implemented was being made we were often asked how the task had to be done. Using the menu to look at how the painting has to be and the quality of our recreation created a discrepancy between what the children expect to see and what is really there in front of them. The task where an object had to be hidden in such a way that it could not see another object also required hints from us. The main confusion came from the task description in the upper left corner, which says "Move the object out of view". During the entire experiment the children bantered a lot with each other, in which we heard what worked in the application and what did not. For the performance of the application a target frame rate of 30 frames per second is the goal.

The questionnaire contained three exam questions, where two of the questions were based directly on the CITO exam questions about perspective and one question was based on one of the tasks present in the application, which was indirectly based on the CITO type questions. The first question tests the children their ability to determine orientation, by having to choose how the scene looks like from a different angle. With the second question the ability to determine whether or not something is visible from a certain location is tested. The second question is the question that is indirectly based on the CITO exam questions and it is the same question that has been made into the position task in the application. The last exam question tests the perspective knowledge of the children based on directions, where they have to choose from which direction the 2D view from an isometric building is made. The full questionnaire can be found in appendix B.

The other half of the questionnaire's questions are subjective. Here the children are asked which tasks they were able to do, which task they found to be the most difficult, which task they liked the most, which version of the position task they found easier, either the one in the exam questions or the one in the application, and we ask them for feedback on the application in the last question.

The children had no trouble with filling in the questionnaire, except for the second exam question for which they needed more clarification on the situation of the task. Some also asked in the first exam question in which direction the character in the figure was looking.

Some children were finished quickly with the tasks in the application and with the questionnaire, while others stayed for the whole hour.

The results gotten from this experiment are explained in section 6.2.

6 Results

6.1 Evaluation Process

After the application is completed and the experiments have been conducted, the application and the results of the experiments are evaluated as per our methodology. The finished application is compared to other similar software. It will be compared to existing software that has 3D reconstructed paintings and their potential to be used for educational purposes. Other software that teach perspective will also be compared to the finished application. The performance of the application is measured with the frames per second present in the application.

Before the comparisons, the results of the experiment are evaluated. The evaluation is done with the help of a few evaluation methods that are based on the methods presented in the overview created by K. Isbister [8]. A questionnaire with questions about perspective and about the application explained in section 5.2, is used to evaluate the before mentioned user experience, ease of use and educational potential. Answers to the exam questions are compared to the previous experience of the children and the experience of the average child according to the CITO exams, this is used to evaluate the educational potential of our application.

Since the group of children that plays the prototype know each other, the children talk about the experiment with each other more often than if it was either played on their own or in a group with children that did not know each other. The banter between the children tells us what they think about the design of the game, this together with the application questions in the questionnaire are used to evaluate the user experience, user engagement and general design of the application. Performing the experiments in groups means that the tasks that the children have to do have more of an emphasize on the game elements. The types of tasks that the children have to do are the same, but each task is randomly generated and has to be fulfilled in a slightly different way. In addition, the questionnaire helps with evaluating the different interactions we created to learn about perspective.

From the questionnaire evaluation of the application and the banter, a comparison is drawn to the other existing software as mentioned before.

6.2 Testing Version 4.0 Results

First, the application itself is evaluated.

On average the frame rate on the used Chromebooks was 22 frames per second. The minimum is 14 frames per second and the maximum 25 frames per second. This shows that the program in its current state is not able to hit the targeted 30 frames per second on mobile software present in smartphones, tablets and Chromebooks.

The banter between the children lets us know what they think of the application. Some had trouble with the position task as to what had to be moved. When placing the objects back to the correct places, they thought everything stood correctly even when this was not the case. The mouse sensitivity on the lowest setting was too fast for 30.8% of the children. Movement controls with the WASD buttons were not natural to some of the children, they used the arrow keys instead.

The exam part of the questionnaire was made with good results, table 3 shows the percentages of correct answers for the children. None of the children had all the questions wrong. A full result

	Percentage of children
All correct	61.5%
Two correct	30.8%
One correct	7.7%

Table 3: Exam questions correct answer ratio

of each children's score with the exam part of the questionnaire can be seen in the appendix section C.

The teacher graded the children's previous knowledge on perspective appropriate to the grade the child was in. The grading was based on a scale of bad, moderate and good. As seen in table 4 most of the children were graded as good, which means they get better results than the average elementary school student. The students from this particular elementary school have more experience with modern technologies like tablets, laptops and other hardware.

	Percentage of children
Good	84.6%
Average	15.4%
Bad	0%

Table 4: Previous experience with perspective

In the CITO scores, the children are placed in percentile groups. The groups go from percentile 10, 25, 50, 75 to 90. Each percentile shows the percentage of elementary school students in the Netherlands that have gotten the score corresponding to the percentile group or lower. The other students not in this group have thus gotten a higher score that corresponds to the higher percentile group. Percentile 50 corresponds with the average student score [14]. The children that participated are either in the 75 percentile category or in the 90 percentile category.

According to the CITO scores, the average elementary school student has a mediocre performance for questions that are similar to the first question of the questionnaire [9]. The 75 percentile children have no trouble with questions one and two, while question three is made at a mediocre level. Performance of the 90 percentile for these questions is the same as the performance of the 75 percentile children.

When comparing tables 3 and 4 with both each other and the CITO scores, it shows that the results are roughly what you would expect that the children can do according to the CITO scores. The difference being that the children that were graded as moderate had successfully completed the task that their percentile group is not good at. The question that they got wrong was the one the average elementary school student is mediocre at.

The amount of tasks the children were able to make is also high, as seen in table 5. There were

	Percentage of children
All tasks	53.8%
Three tasks	30.8%
Two tasks	15.4%

Table 5: Made tasks

no children that only made one or zero tasks. Two of the children that completed three tasks did not finish the position task, while the other two did not complete the placement task instead. The children that completed two tasks did not finish the placement and position tasks.

When asked which task they found the most enjoyable to do, the children were split as seen in table 6. The movement task and position task were both liked equally, while the placement task

	Percentage of children
Tutorial task	0%
Movement task	38.5%
Placement task	23.1%
Position task	38.5%

Table 6: Favorite task

was liked the least. Question two shows that if someone liked the placement task then it is also their choice for most difficult task. In general, as seen in table 7, the placement task was the most difficult for the children to do. If the children are asked which version of the position task they like

	Percentage of children
Tutorial task	0%
Movement task	15.4%
Placement task	69.2%
Position task	15.4%

Table 7: Most difficult task

more, either the paper version or the version in the application, it was a split. The percentages are seen in table 8.

An unexpected result was the amount of information that stayed with the children. 92.3% of the children remembered the names of the landmark point in the reconstructed painting: the Maria church. There is a potential for the application to be used to teach about the painting itself.

	Percentage of children
Paper	46.2%
Application	53.8%

Table 8: Position task version preference

The application from Carrozzino et al. [2] is the closest to what we have created. Their work recreated the painting "The Resurrection" using 3D modeling software and this painting environment was used to enable users to view the painting from different perspectives. However, it was not their goal to teach perspective to the user. They do note in their conclusion that being able to control the viewpoint illustrates the concepts of perspective, showing the potential to use these reconstructed environments for education. Our application continues and expands on this, by focusing on teaching perspective concepts to the user with multiple approaches, from viewpoints to positioning.

An application that is made with the goal of teaching perspective to children has not been made before. This means there are no applications to compare with.

7 Discussion

It is important to note that although the results show a potential for the application to be used as a learning tool for both learning perspective and learning about the painting itself, the amount of children that participated is not high enough to make absolute statements about the application its educational potential.

7.1 Interpretation Of The Results

By analyzing the outcomes of the questionnaire, the assignments showed that the scores were as expected based on the CITO percentile group the children were approximately in. Although the two average elementary school students had the difficult assignment correct and the easier assignment wrong, with the size of the test group it can not be said with absolute certainty if the children learned about perspective with the application. It does show, however, that the application has a potential to be used as an educational tool for both perspective and for knowledge about the painting.

With 69.2% of the children finding the placement task difficult, it could mean that for most reading the depth of an image is hard.

When looking at the subquestions, they have been answered. The types of interactions that have been created based on CITO assignment questions answer what kind of interactions can be used to teach perspective. How the different aspects can be generalized is answered by the information in the implementation section. By using many frameworks that have generic superclasses, it is possible to use the current interactions in any painting. However, creating a new type of task will require the functionality, specific to the new type, to be programmed. The gamification of the environment was a low priority, and ultimately the gamification is put in the task system by having a reward when the placement task is finished. Functionality such as the trapdoors for a reset can also be seen as gamification. The different methods available to create the environment show that manually reconstructing a painting is currently the ideal way to reconstruct it into 3D. The two CITO based tasks show a potential for teaching perspective with the help of dragging and moving of objects and with these interactions the subquestion "what kind of interactions can be used to teach perspective?" is answered. In short, our main research question was how a 3D environment can be created from paintings, which can be used by children to learn about geometric concepts. This is answered by the subquestions.

7.2 Reflection

At the start of the application we were too ambitious with the amount of features we wanted to implement based on teaching perspective and making the application itself engaging for younger users. Some of these are placed in section 7.3 as future work.

Children from different elementary schools and different technical backgrounds may have a different experience than the children that participated.

There are four tasks in the application. However, entirely different types of perspective tasks might give different results.

7.3 Future Work

There are multiple aspects that can be improved. Most importantly, the amount of children to test the application with should be increased. 13 elementary school students is not a large enough pool to make absolute conclusions on what the application can do as a learning tool. However, the main focus here is on how such a tool can be created and when looking at the performance of the application it can still be optimized further. WebGL itself has an overhead for the amount of resources it needs versus a native application. However, in the coming years WebGL will be improved both by itself and in Unity. Currently Unity is beta testing WebGL with WebAssembly.

Another low level optimization coming in the near future is SIMD.js, which allows for Single Instruction Multiple Data operations. These kinds of operations perform a single computation on multiple data points in one CPU cycle. Currently this is only supported in the development builds of Firefox and Edge.

There are more types of tasks that could be made to teach about perspective. The tasks implemented here were the most prevalent ones, but as seen in both CITO exams from 2004 [9] and 2011 [14] there are more types of tasks that can help with teaching perspective. For example, a task where the user has to determine which patterns can create the same object or a task where the user has to select how a person object sees a certain other object.

The feedback from the experiments shows that the children would like to have more paintings to play with. This allows for teaching different geometrical concepts, which in turn allows for different types of tasks.

In its current state the application uses simplistic models. In the future, 3D artists can be brought in to add more details to the reconstructed painting, in that way there is more resemblance between the reconstructed painting and the real painting.

8 Conclusion

How an application can be built from reconstructed paintings and how this environment can be used to teach about perspective have been presented in this thesis. Using a game engine like Unity, with manually reconstructed paintings and using similar techniques from the assignments in the CITO exams an application can be built to teach children about perspective. It can be exported to any platform supported by the Unity game engine, making it available to everyone.

During development of the application, the biggest problem was the performance in WebGL on hardware similar to the Chromebooks the experiments were performed on. We would also have liked to perform the experiment with more children.

With just the group of children that took part in the experiment, we did get some interesting results where the name of the landmark point was remembered by most and where the two average students got the difficult question right. With more time and artists, the design can be improved upon with different task types that follow a different path and better visuals.

Returning to the research questions, what kind of interactions can be used to teach perspective? The types of interactions to teach perspective are currently based on assignments present in the CITO exams which are made to test a child's knowledge about perspective. How can the different aspects in the environment, like gameplay mechanics and objects, be generalized so it can be applied to any 3D environment? In order to keep everything generic, a few abstract classes are created which are used by the systems present in the applications to add functionality and show information. With these classes, elements such as the gameplay and the tasks can be used in any 3D painting that adheres to the implemented tag system. However, if a new task has to be created, then its functionality has to be programmed using the generic task as its superclass. How can gamification be applied on the 3D painting environment for teaching about geometric concepts? In order to keep the application engaging for the user, features such as the trapdoors and the task system are created. These game-elements are used in this non-gaming environment and the tasks themselves are started in an order of least difficult to most difficult. As seen in the results in section 6, the placement task was the most difficult, but for some it was also the most fun task. What

kind of methods can be used to create the environment? Currently, automatic and semi automatic reconstruction methods do not work on paintings, because of an incorrect perspective that is often found in older paintings. This means that in order to reconstruct a painting to 3D and have it be perspectively correct, it will have to be recreated manually using 3D modeling software. With this information a 3D environment can be created from paintings, that can be used by children to learn about geometric concepts.

References

[1] Webgl performance considerations. `https://docs.unity3d.com/Manual/webgl-performance.html`, 2015. [Online; accessed 26-June-2017].

[2] Carrozzino, M., Evangelista, C., Brondi, R., Tecchia, F., and Bergamasco, M. Virtual reconstruction of paintings as a tool for research and learning. *Journal of Cultural Heritage 15*, 3 (2014), 308 – 312.

[3] Christel, M. G., Stevens, S. M., Maher, B. S., Brice, S., Champer, M., Jayapalan, L., Chen, Q., Jin, J., Hausmann, D., Bastida, N., et al. Rumbleblocks: Teaching science concepts to young children through a unity game. In *Computer Games (CGAMES), 2012 17th International Conference on* (2012), IEEE, pp. 162–166.

[4] Cowan, B., and Kapralos, B. A survey of frameworks and game engines for serious game development. In *2014 IEEE 14th International Conference on Advanced Learning Technologies* (July 2014), pp. 662–664.

[5] Elio, R., Hoover, J., Nikolaidis, I., Salavatipour, M., Stewart, L., and Wong, K. About computing science research methodology, 2011.

[6] Faghihi, U., Brautigam, A., Jorgenson, K., Martin, D., Brown, A., Measures, E., and Maldonado-Bouchard, S. How gamification applies for educational purpose specially with college algebra. *Procedia Computer Science 41* (2014), 182 – 187. 5th Annual International Conference on Biologically Inspired Cognitive Architectures, 2014 {BICA}.

[7] Hassner, T., and Basri, R. Example based 3d reconstruction from single 2d images. In *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)* (June 2006), pp. 15–15.

[8] Isbister, K. *Enabling Social Play: A Framework for Design and Evaluation.* Springer London, London, 2010, pp. 11–22.

[9] Jansen, J., van der Schoot, F., and Hemker, B. *Balans van het reken-wiskundeonderwijs aan het einde van de basisschool 4.* Stichting Cito Instituut, Arnhem, 2005, ch. 6, pp. 204–209.

[10] Jansen, P., and Ruttkay, Z. The arnolfini portrait in 3d: Creating virtual world of a painting with inconsistent perspective. In *Annual Conference of the European Association for Computer Graphics, Eurographics 2007 Cultural Heritage Papers* (September 2007), D. B. Arnold and A. Ferko, Eds., The Eurographics Association, pp. 25–32.

[11] Lee, S., Feng, D., and Gooch, B. Automatic construction of 3d models from architectural line drawings. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2008), I3D '08, ACM, pp. 123–130.

[12] Medley, J. Webassembly. `https://developer.mozilla.org/en-US/docs/WebAssembly`, 2017. [Online; accessed 26-June-2017].

[13] Saxena, A., Sun, M., and Ng, A. Y. Make3d: Learning 3d scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence 31*, 5 (May 2009), 824–840.

[14] Scheltens, F., Hemker, B., and Vermeulen, J. *Balans van het reken-wiskundeonderwijs aan het einde van de basisschool 5.* Stichting Cito Instituut, Arnhem, 2013, ch. 7, pp. 254–260.

[15] Sombrio, G., Schimmelpfeng, L. E., Ulbricht, V. R., and Villarouco, V. *Gamification in Education Through Design Thinking.* Springer International Publishing, Cham, 2016, pp. 311–321.

[16] Vouzounaras, G., Perez-Moneo Agapito, J. D., Daras, P., and Strintzis, M. G. 3d reconstruction of indoor and outdoor building scenes from a single image. In *Proceedings of the 2010 ACM Workshop on Surreal Media and Virtual Cloning* (New York, NY, USA, 2010), SMVC '10, ACM, pp. 63–66.

[17] Zhang, Q., Song, X., Shao, X., Zhao, H., and Shibasaki, R. When 3d reconstruction meets ubiquitous rgb-d images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2014).

[18] Zou, C., Peng, X., Lv, H., Chen, S., Fu, H., and Liu, J. Sketch-based 3-d modeling for piecewise planar objects in single images. *Computers and Graphics 46* (2015), 130 – 137. Shape Modeling International 2014.

Appendices

A Code

A.1 A task XML file

This XML file shows the dialogue text from the movement task.

```
<?xml version="1.0" encoding="UTF-8"?>
<task type="movement">
    <title>Verplaats het object</title>
    <description>Zet het object op een andere plek.</description>
    <detailed>Verplaats het object en zet het op de goede plek.</
   detailed>
    <difficulty>1</difficulty>

    <dialogue action="start">
        <variation>
            <line>Oh, hallo.</line>
            <line>...</line>
            <line>Hé, zou je me ergens mee willen helpen?</line>
            <line>Ik probeer iets te verplaatsen, alleen dat lukt
   mij niet. Zou jij dat voor mij willen doen?</line>
            <option>
                <decision action="yes" click="accept">Ja,
   natuurlijk.</decision>
                <decision action="no">Nee, niet nu.</decision>
            </option>
        </variation>
    </dialogue>

    <dialogue action="other no">
        <variation>
            <line>Ah, daar ben je weer.</line>
            <line>Ben je van gedachten veranderd?</line>
            <option>
                <decision action="yes" click="accept">Ja</decision>
                <decision action="no">Nee</decision>
            </option>
        </variation>
    </dialogue>

    <dialogue action="yes">
        <variation>
            <line>Wacht... meen je dat? Super!</line>
            <line camera-view="target">Oké, dit is wat ik wilde
   verplaatsen.</line>
            <line>Als dit nou wordt verplaatst, dan staat het
   volgens mij een stuk beter.</line>
            <line camera-view="goal">Hier is een mooi plekje.</line
   >
            <line camera-view="start">Ik had gehoord dat je dingen
   kunt bewegen met de linkermuisknop <img src="MouseLeftClick"/>,
   ik weet alleen niet wat dat betekent.</line>
            <line>Succes!</line>
        </variation>
    </dialogue>
```

```
    <dialogue action="no">
        <variation>
            <line>Oké. Dat is jammer.</line>
            <line>Als je je bedenkt, weet je me te vinden.</line>
        </variation>
    </dialogue>

    <dialogue action="other yes">
        <variation>
            <line>Lukt het een beetje?</line>
            <line camera-view="target">Met de linkermuisknop <img
   src="MouseLeftClick"/> zou je dit kunnen bewegen.</line>
            <line camera-view="goal">Dit plekje had ik uitgekozen.
   Om het helemaal hiernaartoe te verplaatsen lijkt me toch wel een
    hele klus.</line>
            <line camera-view="start">Maar ik weet zeker dat het
   jou wel gaat lukken!</line>
        </variation>
    </dialogue>

    <dialogue action="finished">
        <variation>
            <line>Kijk, dat ziet er goed uit! Dankjewel!</line>
        </variation>
    </dialogue>

    <dialogue action="other finished">
        <variation>
            <line>Wat is het toch handig dat je dingen kunt bewegen
    met de linkermuisknop <img src="MouseLeftClick"/>.</line>
        </variation>
    </dialogue>

    <floatingDialogue action="start">
        <variation>Hmm, hoe zou het er uit zien als dat daar zou
   staan...</variation>
    </floatingDialogue>

        <floatingDialogue action="other finished">
                <variation>Bedankt voor het helpen!</variation>
        </floatingDialogue>
</task>
```

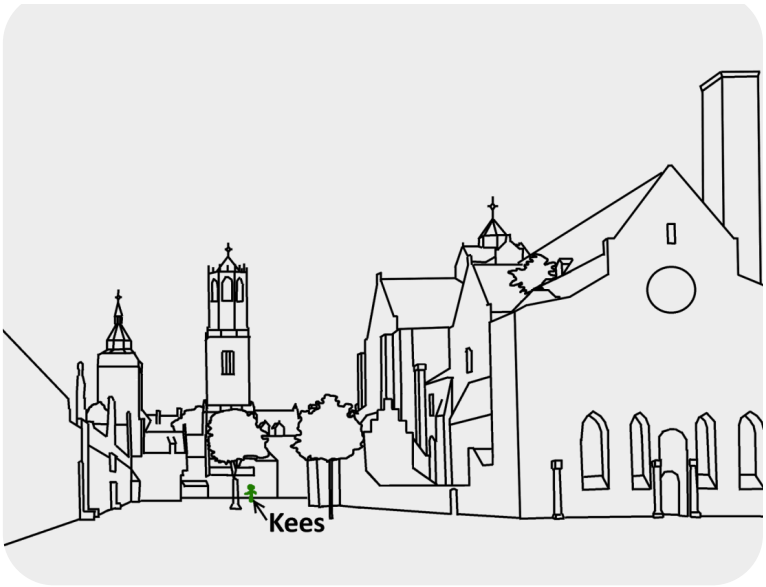
B Exercises and questionnaire

B.1 Exercises

Ik zit in groep: .....

Opdrachten

Ga naar www.students.science.uu.nl/~4007441/Thesis om het spel te starten. Maak daarna de volgende drie opdrachten.



1. Kees maakt nu een foto. Hoe ziet de foto eruit?

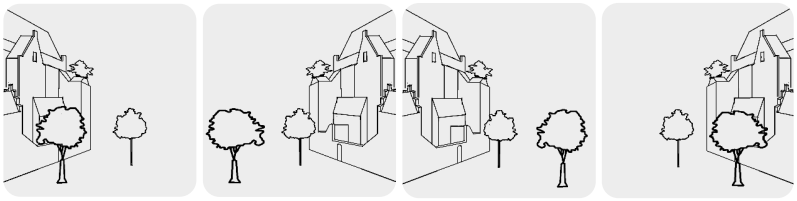
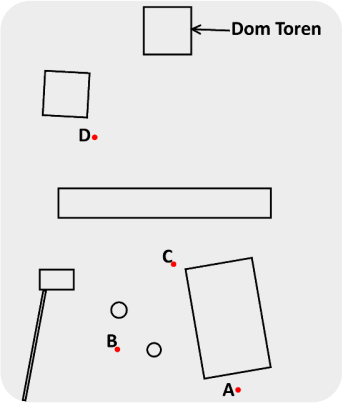


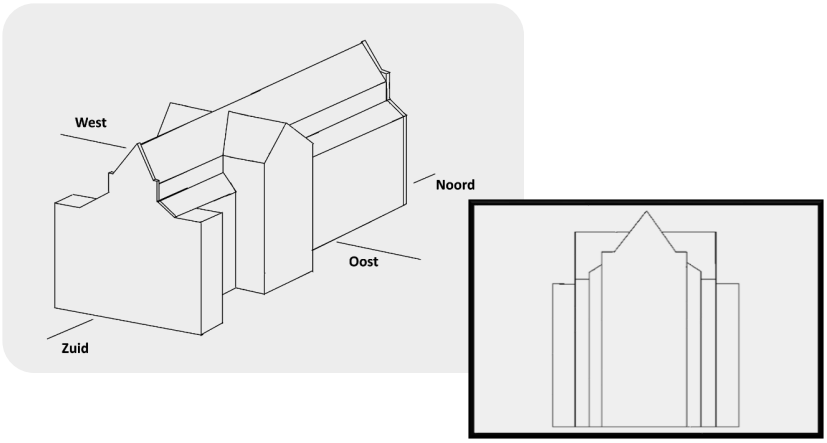
Foto .....



2. Peter kijkt altijd richting de Domtoren. Op welke plek moet hij staan, zodat hij de Domtoren niet meer ziet?

   Op plek .....

3. Hieronder zie je een foto van de Mariakerk. Vanuit welke richting is deze foto gemaakt?



   Vanuit .....

B.2 Questionnaire

Vragen

In het spel kon je vier verschillende taken doen. Kruis hieronder aan welke taken je voltooid hebt. Je kunt in het spel op de tab-toets drukken om te kijken wat de taken waren.

Het museum verkennen ☐
Verplaats het object ☐
Perspectief en gebouwen ☐
Alles staat fout ☐

1. Welke taak vond je het leukst om te doen?

   .....

2. Welke taak vond je het moeilijkst en waarom?

   .....

3. De taak 'Perspectief en gebouwen' lijkt op opdracht 2 van het opdrachtenblad (met Peter en de Domtoren). Vond je de opdracht in het spel of op papier makkelijker?

   .....

4. Wat zou jij willen veranderen of toevoegen aan het spel?

   .....
   .....
   .....

C Experiment results

C.1 Results from the exercises

Child	Experience rating	Exercise 1	Exercise 2	Exercise 3	Score
1	Good	C	A	North	3
2	Good	C	A	North	3
3	Good	C	A	North	3
4	Good	C	A	North	3
5	Good	B	A	Southeast	1
6	Good	C	A	North	3
7	Good	C	A	South	2
8	Moderate	B	A	North	2
9	Moderate	B	A	North	2
10	Good	C	A	South	2
11	Good	C	A	North	3
12	Good	C	A	North	3
13	Good	C	A	North	3

Table 9: The given answers and the total scores from the exercises in the final experiment

C.2 Completed tasks

Child	Tutorial task	Movement task	Position task	Placement task	Total
1	Yes	Yes	Yes	Yes	4
2	Yes	Yes	Yes	Yes	4
3	Yes	Yes	Yes	No	3
4	Yes	Yes	No	No	2
5	Yes	Yes	Yes	Yes	4
6	Yes	Yes	Yes	Yes	4
7	Yes	Yes	No	No	2
8	Yes	Yes	Yes	Yes	4
9	Yes	Yes	Yes	No	3
10	Yes	Yes	Yes	Yes	4
11	Yes	Yes	No	Yes	3
12	Yes	Yes	No	Yes	3
13	Yes	Yes	Yes	Yes	4

Table 10: The completed tasks and the total number of completed tasks

C.3 Results from the questionnaire

Child	Question 1	Question 2	Question 3
1	Position task	Placement task	Application
2	Position task	Placement task	Application
3	Position task	Placement task	Application
4	Movement task	Movement task	Paper
5	Movement task	Placement task	Paper
6	Placement task	Placement task	Application
7	Movement task	Movement task	Paper
8	Position task	Placement task	Paper
9	Position task	Placement task	Paper
10	Placement task	Placement task	Application
11	Movement task	Position task	Paper
12	Movement task	Position task	Paper
13	Placement task	Placement task	Application

Table 11: An overview of the answers given by the children. These questions were asked in the final experiment.