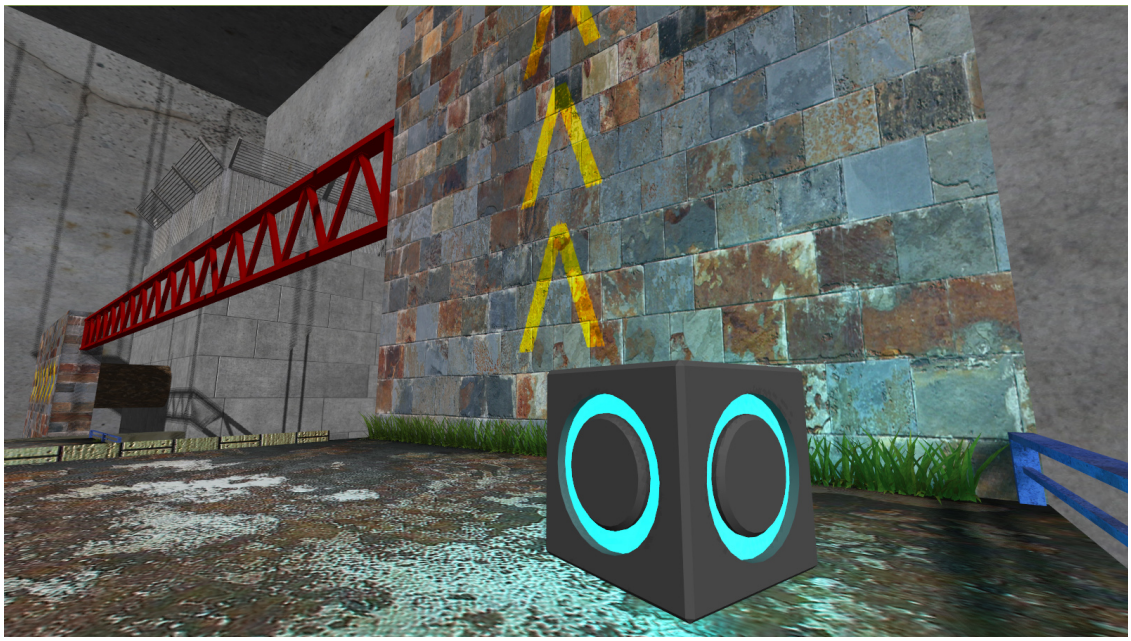


Legend of Cube

Evaluation of the development of a 3D platform game designed for optimal player control

Bachelor of Science Thesis in Computer Science and Engineering

William Dahlberg
Niklas Helmertz
Peter Hillerström
Viktor Karlsson
Christoffer Matsson
Lucas Persson

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2015

Legend of Cube
Evaluation of the development of a 3D platform game designed for optimal player control

William Dahlberg
Niklas Helmertz
Peter Hillerström
Viktor Karlsson
Christoffer Matsson
Lucas Persson

Examiners: Jan Skansholm, Arne Linde

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46(0)31-772 1000

Cover: Image from Legend of Cube showing the main character and player controlled actor in one of the game's levels.

Department of Computer Science and Engineering Göteborg, Sweden, June 2015

Abstract

This thesis explores the subject of game development by detailing the creation of a game. The game is called Legend of Cube and is a 3D platform game. The main design goal was to create a game that let the player feel in control while playing it. How well this is achieved is then evaluated based on user tests.

Techniques in various aspects of game development have been researched and implemented, then assessed through continuous testing during development. This thesis presents concepts like level design, property-centric software architecture, and gives insight into the technology and science behind graphics, physics and gameplay in game development.

The results of our user tests show that the main design goal of the game is, to some extent, reached. We conclude that perfect player control is difficult, if not impossible, and even though the outcome was positive, much more work can be made on the subject.

Sammanfattning

Den här rapporten undersöker ämnet spelutveckling genom att redogöra för utvecklingen av ett spel. Spelet är ett plattformsspel i 3D vid namn Legend of Cube. Huvudmålet var att skapa ett spel där spelaren känner sig i kontroll över spelet. Hur väl detta uppnås utvärderas genom användartester.

Tillvägagångssätt för spelutvecklingens olika delar undersöks, implementeras och utvärderas fortlöpande under utvecklingen. Rapporten behandlar koncept som bandesign men även implementationer och mjukvaruarkitektur, för att ge insikt i tekniken och vetenskapen bakom grafik, fysik och spelkänsla.

Användartesterna pekar på att projektets mål till stor del är uppnått. Att uppnå perfekt känsla av kontroll är svårt, om inte omöjligt, så även om resultaten är mycket positiva finns det stort utrymme för förbättring.

Acknowledgements

We would like to thank our supervisor, Håkan Burden, who was very helpful during the development. We would also like to thank Ulf Assarsson, who gave advice about what tools to use.

Contents

Glossary

AABB - Axis-Aligned Bounding Box

BV - Bounding Volume

C# - A programming language which can be compiled to bytecode, running on the CLR

CLR - Common Language Runtime, a virtual machine for running programs written in various Microsoft languages

Direct3D - Microsoft API for 3D graphics

DirectX - Collection of Microsoft APIs for multimedia applications

Game engine - Frameworks which provides various functionality needed for a game.

Gamepad - Type of controller held in two hands with two analog sticks, designed for video games

Legend of Cube - The game that has been developed for this thesis project

Level - A virtual space filled with obstacles and objectives, and in which the gameplay takes place

LoC - See Legend of Cube

OBB - Oriented Bounding Box

Platform game - A game genre characterized by levels that involve jumping between platforms and over obstacles

Platformer - A game in the platform game genre

Player - The user playing the game

Player character - The in-game avatar that the player controls

XNA - A game development framework created by Microsoft

1
Introduction

Video game development is a continually changing landscape. Recent years have seen a rise of independent developers, often shortened to *indie developers*, and refers smaller studios not backed by a video game publisher [1]. Indie developers have gained a reputation for experimenting and creating games that traditional publishers will not.

Platformers have seen a resurgence through this rise of indie developers. Platformers, or platform games, is a type of video game where the focus is on the player character's movement mechanics themselves, usually manifested by having levels in the form of obstacle courses that the player needs to traverse. The genre was popular during the 2D era, but interest among larger developers lessened with the transition to 3D. Platform games in 2D is however a common genre for indie developers, with an example being *Super Meat Boy* [2].

Even though 2D platformers are once again common, relatively few platformers in 3D are released. One explanation might be that getting the platforming mechanics to work satisfactory in 3D is complicated. It is therefore of interest to investigate the process of creating a 3D platformer in order to understand the various development challenges that arise.

1.1 Purpose

The purpose of this thesis is to detail and evaluate different aspects of the development of a 3D platform game. Additionally, this thesis evaluates one particular design goal of the game, which was to make the player feel *in control*. What is meant by *in control* is defined in section 1.2.

1.2 Problem Statements

Developing a game requires solving problems is many different areas. In this thesis, the development has been divided into five problem areas, which are handled separately. What is to be answered in each is the following:

- *Gameplay*: What mechanics the game has, and how they are implemented.
- *Level Design*: What techniques that have been used when design of the levels.

- *Software Architecture*: What architectural techniques the developed game has been based on.
- *Physics*: In a realistic simulation of a 3D world, there is the need for simulating physical laws. How this is done in the game, for enabling the chosen game mechanics, is to be answered.
- *Graphics*: A 3D game needs consideration of graphical appearance. The thesis will answer how the camera works as well as how the game world is rendered.

Aside from these five problem areas, there are some other general goals of the project – one of them is to keep the hardware limitations to a minimum. Today, developers are competing in making the most impressive and realistic looking game – increasing the demand for more powerful hardware. Having to upgrade components in order to play the latest games, adding to the ever growing problem of e-waste, is not sustainable. Thus, to not add to this issue, Legend of Cube is aimed to function even on low-end computers.

As previously mentioned in the purpose, the thesis aims to explore how to make the player feel *in control*. In control is, for the purpose of this thesis, divided into four key aspects. These, and their definitions, are:

- *Consistency*: The same input and situations should yield the same results. The game should never surprise the player by acting differently in a specific situation.
- *Predictability*: The player should be able to predict what is going to happen next and not be surprised by unexpected elements.
- *Freedom*: The player should be able to control the game as freely as possible and not feel constrained. I.e. the player should be able to choose different approaches without feeling that the game forces them into playing the game a specific way.
- *Natural feeling*: The game and controlling of the character should feel and look intuitive and natural to the player.

1.3 Scope

We chose to focus more on having good gameplay, see chapter 3, and only implement the most essential features in graphics, physics and level design to achieve our goals for the gameplay. Additionally, the plan from the start was to have only one level that would highlight our game's features as well as possible.

The game engine in Legend of Cube was largely written by us instead of using an existing solution. This was done so that more aspects of game development could be explored. Creating the game from the ground up allowed for exploration of the software architecture of a game engine and how the various parts affect each other. Due to this decision, some things had to be excluded from the project in order to be able to finish a playable prototype.

During the graphics development, focus was not on implementing particularly advanced graphical effects. It was thought they would take too much time to implement – time that could be better spent improving the camera mechanics. However, one effect that was still considered important is shadows, as they are important for judging distances and depth in a platform game.

1.4 Outline

The report is structured to give the reader an accessible way to read about a particular aspect of the developed game. This is done by dividing the thesis into a number of chapters, each for a different domain, that all contain some result and discussion, rather than having a single larger discussion chapter toward the end.

To illustrate the outline, take chapter 3 on gameplay. It starts by presenting the general idea of the game, which is followed by sections on different gameplay mechanics. Each of these start by presenting background, and is concluded with result and discussion. The following chapters then follow a similar pattern, but cover other aspects of the game.

The initial chapters of this thesis are on topics related to user experience, like level design, which is followed by chapters on subjects that are more technical. This is to give the reader a more general idea of what functionality the game has, before focusing on how it is accomplished.

2
Method

Developing a game requires decisions on several aspects of the process, ranging from what framework to use, to how high the character should jump. These decisions have largely been made through trial and discussion within the development team. In the following sections, we detail the process used during development, such as what initial technical choices were made and how the game has been evaluated.

2.1 Development Process

At the beginning of the project, some decisions about how we were to work on the project were taken. Generally, we wanted to work in an iterative manner, rather than fully defining the requirements at the very start. However, the workflow did in many ways evolve over time as we discovered what worked and what did not.

2.1.1 Backlog

A backlog was used in this project to keep track of what needed to be done and to see what other members were doing. Weekly meetings were held where the backlog was reviewed. On these meetings, it was decided which backlog entries were most important and which project member should deal with which task. If a project member was done with his assigned task, or did not get any specific task assigned, he could always check the backlog for what to do next. Using this method we could assure that there was always something to be done.

2.1.2 Version Control

Github[1] was used for version control during the project. Whenever a project member started on a new feature, it was decided that they should create a new branch, and work solely on that feature there. When a feature branch was completed, a *pull request* for it was opened, where the code would be reviewed by other members to ensure correctness and code quality, before being merged into the development branch.

2.2 Development Tools

It was decided that Microsoft's XNA framework [3], along with the C# language, would be used for the project. The main reason was that XNA seemed to provide

[1]Online Git repository hosting service, found at `https://github.com/`

a fair bit of functionality, but still gave a substantial degree of freedom in how to implement the game, as opposed to most complete game engines. It provides a basic game loop, audio, input and window management, among other functionality common to games. An advantage of using the C# language is that it is similar to what most group members have worked with previously.

The game targets only the Windows desktop platform. Cross-platform development with XNA is technically possible, with the help of Monogame [4], but it was still excluded in order to avoid spending time on something that could easily turn out to be more time-consuming than expected.

All 3D models were created using *3DS Max 2015*, a 3D modeling software created by Autodesk Inc. and all sound effects were designed, mixed and mastered in *Ableton Live 9*. The choice of modeling software and digital audio workstation were done based on personal preference and previous ownership and experience of the products.

2.3 User Testing

User tests were conducted in order to evaluate the gameplay with regards to consistency, freedom, predictability and intuition. A total of twelve persons participated in the tests, four of which had little to no experience playing video games. All tests started by handing a gamepad or a keyboard to the participant. Out of the twelve participants, six used a gamepad and the others a keyboard. Upon receiving the input device, the participants were told to try to complete "Level 1".

The tests were all performed on the same computer, a machine without loss in performance or quality. The game was played with the same, well-working, gamepad by each user. The users were 85% male and around 50% considered themselves experienced using a gamepad to play games.

3

Gameplay

> *"A game is not defined by its technology but by its gameplay. Gameplay can be defined as the overall experience of playing a game. The term game mechanics pins down this idea a bit more concretely - it is usually defined as the set of rules that govern the interactions between various entities in the game."*
>
> – Jason Gregory [5, p. 847]

This chapter presents the game itself by describing the idea behind the game and how it is played. Additional information on the game can be found through its official web site[1].

The main goal of the game's design was to give the player control. The movement should enhance the players feeling of being in control. This is of course a very subjective goal; feeling good might differ from person to person. We have tried fine-tuning the movement of the character, and later performed user testing to evaluate to which extent our goals have been fulfilled.

3.1 Gameplay Concepts

Legend of Cube is a platform game set in a 3D environment. The goal of the game is to navigate the game avatar, the cube, through levels with an emphasis on doing so as fast as possible. The game is intended to be played with a gamepad (most preferably an *XBOX360 controller* [6]) but a keyboard can also be used. The gameplay is designed to be fast-paced and usually straightforward when it comes to finding the way. The game is played in different levels (see Chapter 4), which are to be traversed by the main character: the cube. The cube has a few abilities; it can slide on the ground; it can jump with variable power, bringing it to different altitudes; and it can slide along and jump from walls (which can be seen in Figure 3.1 and 3.2). These mechanics are what distinguishes the gameplay, and are the core techniques that a player will have to master to complete the game.

[1] See `http://legendofcube.github.io/`

Figure 3.1: In-game screenshot the player cube sliding along a wall.

Figure 3.2: In-game screenshot of character just having jumped off a wall.

While the main challenge is to be able to traverse the levels and get to the finish line, a rather large part of the game is to complete the levels as quickly as possible, which can be done by optimizing the choice of path to the goal, and fine-tuning the execution of the planned path. This concept is called *speedrunning* [7] and might increase the replay value, which is the potential satisfaction of playing the game again after finishing it.

3.2 Jump Mechanics

In every game that involves jumping, there needs to be a defined model for how the character behaves while doing so. Different games might benefit from different models. Certain kind of games, such as Legend of Cube, which is strongly focused around jumping and moving, need more advanced and thought-through jump

mechanics. This section will cover the important aspects when characterizing a jump; how it is solved in some popular games, and finally, how the problem was approached in Legend of Cube.

3.2.1 Aspects of Interest

In real life, it is not possible to control one's direction much while in air, with some exceptions in the animal world of course. In games, the approach is a lot less scientific, to allow for better gameplay. Thus, is it often possible to change direction while in air, even though there is no anchor in reality for doing so. This is the first aspect given special consideration, while studying jump mechanics.

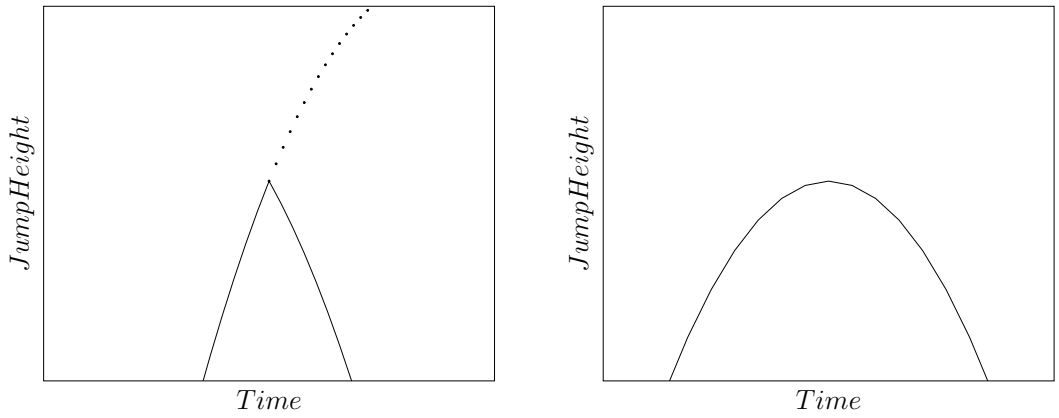
A more realistic example and the second aspect we have analyzed is the ability to jump variable heights, which is desirable, since you do not necessarily need a jump of maximum height to clear a certain obstacle. Furthermore, variable jump height can positively affect the level design, since it adds another concept of challenge, where good jump control is needed to pass some part of a level. There is more than one way to implement variable jump height, and how the implementation affects the player will be discussed in the following section.

The third aspect that was chosen to be analyzed more deeply, that is present in many popular platforming games, such as *Super Meat Boy* [2], *Super Mario 64* [8], *Super Mario Sunshine* [9] and *Speedrunners* [10] is wall jumping. Jumping up to a wall and then jumping away from it can be a good way to clear obstacles and traverse forward in the level.

3.2.2 Research

We have looked at games such as the above-mentioned Super Mario games and Super Meat Boy (abbreviated as SMB) and analyzed their jump mechanics, with a focus on the previously mentioned aspects.

There are two main ideas on how to implement a variable jump height. In Super Meat Boy, you hold down the jump button until you have reached the desired height and then let go. As the developer described, the player character then immediately loses all upwards velocity and starts falling [11]. In recent Super Mario Games, more specifically Super Mario Sunshine, you vary Mario's jump height in the beginning of the jump by either tapping for a short jump or pressing a bit longer for a high jump. Figure 3.3 describes the differences of these models.



(a) An interrupted jump in Super Meat Boy. The graphs describe the complete behavior of the jump, while the solid parts are what actually is executed.

(b) A jump of any height in Super Mario Sunshine.

Figure 3.3: Visual representations[2] comparing the jump models of Super Meat Boy and Super Mario Sunshine.

The advantage of SMB's model is a very precise and responsive jump height. A drawback, however, is that the game cannot predict when the player will release the jump button. This introduces difficulties in determining the character's jump trajectory and may result in bad jumping animations. SMB solves this with a direct approach, as seen in Figure 3.3a, with a broken curve. Super Mario Sunshine has a visual advantage in this subject due to the jump height being determined before the character reaches the minimum jump height. This allows for a continuous jump curve, which is more aesthetically pleasing; without regards to such curves is this, however, a less precise way of dealing with the ability to vary the character's jump height. What is desired will vary on the amount of precision needed in the game.

In SMB, the air control is high, to the point that the player can make a jump full speed in one direction, and then change direction in air and land at the point where the jump started. Such high in air control puts pressure on the level designer since it could potentially break the level design and make it too easy. This did however does not appear to be a major issue, neither in our, nor other consumer's opinion, based on Steam[3] reviews [12].

Due to the high in air control in SMB, it is possible to perform multiple wall jumps against a single wall, this is not the case in Super Mario Sunshine where the air control is much lower, to the point where wall jumping is only possible against opposite walls.

[2]These graphs are not the actual functions of the jumps, just a visual aid aimed to explain the differences between them.

[3]Steam is a popular platform and community for buying and playing games online, `http://store.steampowered.com/`

3.2.3 Results

Much play testing and tweaking was made, in order to find the final model of jumping that is present in Legend of Cube; the jump we ended up with is a mix of the two jumping models studied in 3.2.2. The core jump is closer to Super Mario Sunshine's implementation, where the jump height is calculated early in the jump, based on how long the initial jump input was; this gives a more arced jump but a little less precision. The amount of air control and wall jumping behaves more like in Super Meat Boy; the wall jump is calculated in a similar way as the normal jump. The ability to jump several times from the same wall is limited but possible. While we do not have total air control as in SMB, we still have enough so that the player is able to completely change direction in air.

3.2.4 Discussion

Early on in the project, we did not have a clear model of jumping, which caused many problems, since it was difficult to fine-tune the jump without a real model. When we reworked the jump into the one we currently have, we wanted a jump where we could calculate the jump height before landing, but still have some variability in it. We realized that, while this method sacrifices some precision, it gives a more natural feeling and jumps that are more consistent.

The amount of in air control early in the development was high. It was later limited, since it was possible to climb a single wall by repeatedly wall jumping on it, which could break the level design. It is still possible to a small extent, it is however very slow, and needs perfectly timed jumps every time to reach any elevation. This was deemed not an issue with the approach that LoC should be speedrunner friendly.

Strafe jumping and bunny hopping has been around for a long time in many games based on the Quake-engine, these jumps are both described in *SourceRuns* wiki [13]. As mentioned in the wiki, it was first discovered as a bug and it is a way of jumping and strafing to increase the characters speed above the normal max speed. It is a way to perform speedruns (see 3.1) or to just move exceptionally fast in the affected games. A similar concept appeared in LoC when implementing wall jumps. When a wall jump is performed, the character instantly receives some velocity to move away from the wall. This addition of velocity made it possible to travel faster than the usual maximum speed. The feature was kept to encourage the possibility to do speedruns in LoC.

3.3 Movement

The movement system might be easy to overlook or take for granted while looking at games, even in platform games where the movement is integral to the gameplay. This section will describe how we implemented the movement system and why we made the choices we did.

3.3.1 Results

We designed the movement to make the cube feel heavy, invoking a feeling of inertia; this was done through having accelerations. Every time the cube starts from a complete stop, it takes a short while before it reaches its maximum velocity, and likewise when it decelerates to a stop. The direction of the cube is also directly related to the direction of the camera, as mentioned further in section 7.2.2. A side effect of this is that a manual rotation of the camera, while moving forward, causes the cube to turn in the direction that the camera is rotating. We designed Legend of Cube with a gamepad in mind as the main way of controlling the character; this enables us to be able to take analog input from a stick varying from 0.0 to 1.0, which makes it so that the cube can move slower or faster depending on how tilted the stick is.

3.3.2 Discussion

The design choice to include inertia in Legend of Cube enhances the natural feeling of the game, but comes at the price of precision controls; for example, stopping at an edge becomes much harder. The other way of doing it, with instant stopping and starting, does not work as well for platform games in general, since many such games build upon the mechanics to preserve movement velocity to overcome obstacles.

4

Level Design

The designing of levels is one of the more important elements when creating a natural gameplay. Physical constraints, such as jump height, and the skill progression of the player, are two examples of what the level designer must take into account when creating a level that is fun and rewarding to play. For the game to be fun, it is usually good to think about fairness to the player too. That is, make sure there are no invisible traps that the player does not understand, but still provide a challenge, facilitating the feeling of accomplishment for the player, upon level completion. This section will go into a bit more depth on what is good and bad when it comes to level design.

Level design is a broad term and can refer to different concerns depending on which game or genre that is being discussed, generally it refers to the creation of levels, maps, missions, environments and other spaces which the player can explore. When designing levels in a platform game, such as Legend of Cube, there are a few ideas to keep in mind, which is what will be discussed in this chapter.

4.1 Level Design in Platform Games

When designing platforming levels and single player games in general, it is important to have good "gameflow", as Bleszinski [14] calls it. He refers to it as the life force that makes a good game fun, and that it is mostly about the reward-response system, which challenges the player and then rewards them for completing tasks. The point is that the Gameflow is completely in the hands of the level designer, since it is the level designer's task to create the challenges and the following rewards for motivating the player.

Since platforming games are essentially obstacle courses where you need to reach the end, it is good if the levels are quite linear and have some kind of flow that leads the player, so that the player does not get confused on where to go [15, p. 110]. It can however be good to give the player some kind of freedom on where to go or at least give them the illusion of freedom, this can be done through creating multiple paths where the player can choose which path to take, even though they may all lead to the same goal [14]. The visual design is important when creating a level with good flow, giving certain platforms special colors or even just putting up some simple direction arrows can help indicate where to go next. An example of this is the game Mirror's Edge [16], which has mostly open levels but they use stark colors to indicate obstacle and doors that you can interact with, which helps the player know where to go.

Another important thing about level design is to ease the player into new type of obstacles, so that at each obstacle all the necessary information has been provided beforehand [17]. An example of this, in LoC, would be a part of "Level 1" where the player is supposed to jump between three walls, where earlier in this level the player has been introduced to walls which you can jump away from.

4.1.1 Results

When the levels in LoC was designed we did not have much knowledge about designing levels except for our own experiences from playing similar games, so the design process was very much trial and error trying to find what was considered by us to be a fun level. It is also important to notice that the creation of levels had a very low prioritization throughout the project, and our initial goal was to have one level that highlighted most of the games features. The result was one complete level and several test levels, which were produced during the development for testing different features. The complete level was found by both developers and testers (see section 8.1) to be a quite fun level, there was a small contest among the developers to see who could complete it in the fastest time.

4.1.2 Discussion

There is a multitude of decisions we might have made differently, if we had had more time and experience to work with our levels and their design. The one thing we would have liked to do, which would have directly impacted our game, would have been to make a tutorial level to introduce the controls and all of the game mechanics to the player. Unfortunately, it was low priority and we did not have enough time. A big reason we decided not to create more levels was that it was a very time consuming process, since we neither had a level editor, or a level format. This meant that we constantly had to guess the placement and manually test the level between every adjustment made to the level. Furthermore, since our purpose relates entirely to the control and movement of the character, only one level was actually needed to assess the smooth gameplay we were striving for.

4.2 Learning the Game Through Level Design

A hallmark of good level design is to have the level teach the mechanics of the game to the player without having long tutorial messages pop up on the screen; for example, the first level in LoC starts the player off on a lonely platform with the only way forward being another platform a small distance away. Most players realize that there should be some way to make it to the next platform and a common reaction should be to try to jump there, then suddenly the player has learned to jump without the game telling the player how to do it. After that, the next platform is even further away and if the player attempts to make a normal jump there he will not reach it, but since there is no other way forward the player might realize that if he holds the jump button down, the character makes an even higher jump. Using this method to learn the game, it can be more interesting to players, since

they learn to use the mechanic, in this case jumping, in an ordinary game situation [18, Chapter 2.3].

4.3 Puzzle Versus Skill-based Platform Games

There might be considerable differences in level design, even within the platforming genre. There are puzzle platforming games, which usually offer the player a calm progression. Popular titles include Braid [19], Portal [20] and Thomas Was Alone [21]. In puzzle games, the finish point of the level, and the way to get there, is mostly not very clear. The purpose of such a game is to challenge the player mentally. The player is supposed to scrutinize the level for clues, or use a different approach or technique to approach the problem, finally finding a way to the finish line. These kind of levels require attention to detail; it is important that objects that are important for the player, are possible to find, and that the clues are reasonable enough. Guesswork is, of course, part of the puzzle, but when it becomes far-fetched, it is possible that the player loses interest.

Another type of platforming game, that requires another type of level design, is the action platforming game. The levels in these games often have a clear path to follow. Instead, the challenge lies in the physical skills of the player. Dexterity and reaction time play an important role, and the game is usually not mentally challenging. Legend of Cube is good example of such a game, where it is more important with the larger composition of the levels. Instead of detail in the tiny things, that might hint to a solution, you mostly already know where to go, and how. The problem here is to actually manage to do it, only using the mechanics described in Chapter 3, thereby refining your skills.

4.4 Method of Research

The level designing has been tested in several different levels during the development of LoC, but the main testing method has been “Level 1”, seen from above in Figure 4.1. Level 1 utilizes several of the techniques described in the previous sections of this chapter, such as learning through the level design, usually with only one way forward, forcing the player to learn and perfect their ability to control the character. Another important aspect when designing a level is of course to take into account the jump height and jump length or the character.

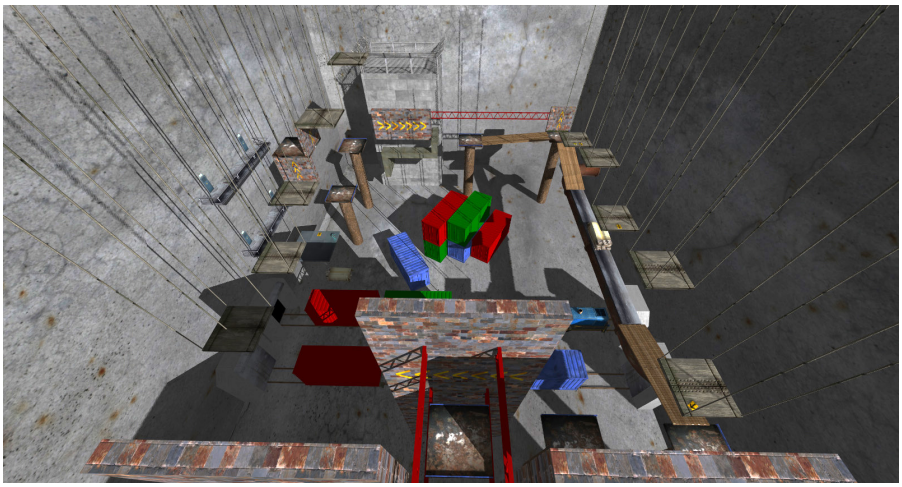


Figure 4.1: In-game screenshot of “Level 1”, the primary level of the game.

5

Software Architecture

The software architecture of a game is often split into two parts, the *game engine* and the game itself. A game engine can be seen as a collection of components, such as a rendering and physics component, on which a game can run [5, p. 11]. The game itself is then the art assets and logic needed to run the game. In the early days of game development there were no separation between these parts and games were typically completely written from scratch every time [22], and even today the separation can be quite blurry [5, p. 11].

One of the more important jobs of a game engine is to keep track of all the objects in the game. Examples include the player character, enemies and items. Common names for these objects are entities, actors and agents [5, p. 853]. The part of the engine responsible for these objects is sometimes referred to as the *gameplay foundation system*; this system will be covered in section 5.2.

A game engine was developed for Legend of Cube. It is quite strongly coupled to the game logic, since it was never intended to be reused for other games. It was however intended to be *data-driven* in order to facilitate user-created levels. It uses a *property-centric* gameplay foundation system in order to accomplish this. This chapter will mainly cover this game engine and some other interesting aspects of the software architecture in Legend of Cube.

5.1 Data-Driven Engines

An important concept when talking about game engines is whether the engine is *data-driven* or not. In a completely data-driven engine, the game is defined in external scripts, configuration files and media assets [22]. In a non-data-driven engine on the other hand, part or all of the game is defined in code.

There are advantages of having a data-driven engine. Since the game is largely defined in external files, game designers and artists do not need to touch the code when creating the game [22]. Another advantage is that it could allow the users to modify the game and create their own content [5, p. 11].

5.1.1 Discussion

One of the early intents with Legend of Cube was to define levels completely in external files and therefore make it somewhat data-driven. At the very least it would probably make it easier to create levels and level creation tools. It could also potentially have allowed for user-created levels.

Unfortunately, this was never accomplished. However, the infrastructure to implement it is partially in place, in the form of a property-centric gameplay foundation system, which is described in more detail in section 5.2.

5.2 Gameplay Foundation System

The *gameplay foundation system*, as described by Gregory [5, pp. 869-891], is responsible for keeping track of objects in the game. He goes on to describe two common architectural styles for representing game objects, *object-centric* and *property-centric* architectures.

The object-centric style can be seen as the standard object-oriented programming approach. Commonly, each type of game object has its own class and each object is represented by a single instance. There is usually some sort of inheritance structure in place.

The property-centric style instead focuses on the properties of the game objects. Typically, a game object is just represented by some sort of unique identifier. Each property is then just data or an instance of another class stored in separate collections with the identifier as a key. What an object actually is, is then defined by what properties it has.

```
struct Entity {
    Vector3 position;
    Vector3 velocity;
    Vector3 acceleration;
    Model model; // 3D model
}

// array-of-structs (list of game entities)
Entity[] entities = new Entity[numEntities];
```

Listing 5.1: Example of array-of-structs layout in C#.

```
struct Entities {
    Vector3[] positions = new Vector3[numEntities];
    Vector3[] velocities = new Vector3[numEntities];
    Vector3[] accelerations = new Vector3[numEntities];
    Model[] models = new Model[numEntities]; // 3D models
}

// struct-of-arrays (list of all game entity properties)
Entities entities = new Entities();
```

Listing 5.2: Example of struct-of-arrays layout in C#.

The difference between object-centric and property-centric style can be compared with the difference between the *array-of-structs* and the *struct-of-arrays* layouts. Listing 5.1 and 5.2 shows the same simplified gameplay foundation system using

array-of-structs and struct-of-arrays respectively. In the first example, an entity is an object while in the second it is an index.

A problem with the object-centric style is that it is hard to define new types without changing the code [22], as exemplified in the following quotation:

> "Initially the development started under the impression that the non-player characters (NPCs) and doors do not share many components. Later on, the designer wanted to allow the player to have a conversation with a door just as he can have a conversation with NPCs but since the initial design only allowed NPCs to have the conversation component, they found that pushing the component up the hierarchy was very difficult and resolved to use a hack around the problem." [22]

In contrast, property-centric style makes it is very easy to create new types in data. If a new object is to be created there is no need to create a new type of class, all that needs to be done is to create a new entity id and add the wanted properties.

5.2.1 Results

In Legend of Cube there are no static objects, everything is handled by the gameplay foundation system and considered a game object. Walls, platforms, the player character, decorations, spawn zones, etc., are all entities. This approach is used for its implementational simplicity.

The gameplay foundation system is implemented in a property-centric style. Each entity has a mask defining which properties it has. Listing 5.3 shows how such a mask is defined, each bit in the integer is a boolean value representing whether the entity has some specific property or not.

```
const UInt64 NO_PROPERTIES = 0;
const UInt64 POSITION = 1 << 0;
const UInt64 VELOCITY = 1 << 1;
const UInt64 ACCELERATION = 1 << 2;
...

// Can combine multiple properties with bitwise OR
UInt64 posAndVelMask = POSITION | VELOCITY;
```

Listing 5.3: Simplified example of how the property masks are defined and can be combined.

Each entity's mask and all property data is stored in a central class called *World*. The masks and the property data itself is stored in equally sized arrays, i.e. the struct-of-arrays layout. An entity is then simply an index, which can be used in all these arrays, only the data available according to the property mask is valid. Listing 5.4 shows a simplified example of the World class.

```
class World {
    UInt64[] propertyMasks; // All entities has a mask
    Vector3[] positions; // POSITION
    Vector3[] velocities; // VELOCITY
    ...
}
```

Listing 5.4: Simplified example of how dynamic entities are stored in Legend of Cube

A consequence of this pattern is that logic is no longer being applied to specific types of objects. Rather logic is being applied to all entities in the world, which have some specific properties. Listing 5.5 shows an example of how the position of all entities with a velocity can be updated.

```
UInt64 posAndVelMask = POSITION | VELOCITY;

for (uint i = 0; i < world.NumEntities; i++)
{
    // Bitwise AND to mask out the specific property bits
    // then compare with precalculated mask to see if
    // entity has the needed properties

    if ((world.propertyMasks[i] & posAndVelMask) ==
        posAndVelMask)
    {
        // Updates entity's position
        world.positions[i] += world.velocities[i] * timeDelta;
    }
}
```

Listing 5.5: Example of how a property mask can be used to update the position of all entities with both the position and velocity property.

With this structure, it is very simple to redefine what a specific object is. Take a stationary wall for example. By simply adding the *VELOCITY* property to its mask and setting a velocity the wall will start to move.

5.2.2 Discussion

The group found that using the property-centric system felt very unusual at first, but it felt very flexible and powerful after people had gotten used to it. It was convenient to not have to create a new class every time a new type of object was created. It was also very useful to be able to quickly add properties to an existing object if it was missing something.

In a sense, you are doing memory management on your own, as data can be accessed even if it is not valid. At the start of the project, it was feared that this would become a problem. In practice, this did not happen very often, and when it did, it

was quickly found. We theorize it might become bigger problem in a larger project with more people involved.

The property-centric architecture will likely make it easy to convert the engine into a data-driven one. It is trivial to design a system where you choose properties and values in an external format, using for example XML. That is really all that is needed to create all kinds of objects supported by the engine.

5.3 Performance

Within the game industry, performance is often of high priority during development. Mike Acton, engine director at Insomniac Games, claims that performance is so important that if it was feasible, they would work only in assembly languages [23, p. 7]. Similarly Jason Gregory, lead programmer at Naughty Dog, claims that games need to perform as efficiently as possible [5, p. 152].

As a consequence of this need for performance, a concept called *data-oriented design* has become common. In a talk at *CppCon* [23], Acton describes it as a different way of thinking about programming, where data and how it is laid out in memory is more important than the code itself. The motivation behind it is that the primary bottleneck in modern systems often is bandwidth and memory latency, not computational power.

5.3.1 Results and Discussion

Whenever possible, contiguous memory allocation is used. The most common data structure used in Legend of Cube is arrays of value types. Value types are allocated on the stack in C#, which means that the arrays will be contiguous in memory. This alone hardly makes the game data-oriented, but it is still worth mentioning.

Data-oriented design was generally considered out of scope for this project. This was due to three reasons. Firstly, it would take too much time. Secondly, it was thought that C# lacked the amount of control needed to make Data-Oriented programming work well. Thirdly, Legend of Cube is quite small in scope and it was assumed that performance would not turn out to be a major problem. This assumption held, even if some things turned out to be more expensive than they optimally needed to be.

5.4 Stuttering

According to Gregory, a video game is often considered a soft real-time system [5, pp. 9-10]. A real-time system is a system with hard deadlines, i.e. limits indicating the latest time a task may finish. A video game is a soft real-time system since it is unlikely that a missed deadline will be fatal. An example of a deadline, mentioned by Gregory, is the need to render to the screen at least 24 times a second, otherwise the objects will not look like they are moving properly.

A potential problem when trying to meet the aforementioned deadlines is non-deterministic delays. An example in C# is the garbage collector that runs spontaneously when the *CLR* (Common Language Runtime) decides it is necessary, which leads to the program execution stopping for a short while until it is done [24]. This introduces a short unpredictable delay into the game, which causes stuttering, i.e. the game temporarily stops for a short while.

5.4.1 Results

Dynamic memory allocation is avoided during runtime in order to avoid spontaneous garbage collection. The dynamic memory needed during a session is allocated once when starting the game and then reused. This has some consequences, for example, the amount of game objects that can exist at the same time is set and can not be increased during runtime. For temporary variables, classes are avoided in favor of structs, which are usually allocated on the stack and therefore not garbage collected.

5.4.2 Discussion

No significant stuttering has been observed in Legend of Cube. It is therefore assumed that the measures taken to prevent this from happening have had an effect. The primary measure, not allocating dynamic memory during runtime, was not very difficult, thanks to the architecture of the gameplay foundation system (covered in section 5.2).

6

Physics

For the gameplay of a platform game to function, the player character needs to be able to traverse the world and collide with objects, such as walls and floors. These features can be achieved using some sort of physics simulation. Legend of Cube uses a very simple bare-minimum physics engine, in order to make the desired gameplay possible. This chapter covers the basic aspects of this physics engine.

6.1 Physics Simulation

In his book, *Game Engine Architecture*, Gregory suggests [5, p. 651] that there are three design impacts to consider before adding physics to a game. They are as follows:

- Predictability: It can be hard to predict how a physics simulation will behave.
- Tuning and control: The laws of physics are fixed. It can be harder to control and make something behave in a specific way.
- Emergent behaviors: Unexpected behaviors can emerge due to the unpredictable nature of physics.

Besides these design impacts, there is also a choice to be made between developing a custom physics engine, and using an existing one. According to Gregory, [5, p. 653] writing a physics engine is both time-consuming and challenging.

6.1.1 Results

A bare-minimum physics engine was developed for Legend of Cube. This was motivated by concerns that it might be more difficult to properly explore various platforming mechanics, such as jumping, with an existing solution. I.e. a custom solution would be more open for modification and control, to enable specifically the kind of gameplay wanted. In order to avoid spending too much time, its implementation was kept as simple as possible.

The choice to have physics in the game was made because it was thought necessary in order to enable the development of responsive and good jump mechanics. There were some concerns over the predictability aspect of the physics simulation, platforming mechanics need to be predictable and behave exactly as the player expects. Except for some minor edge-cases, this luckily enough never turned into a problem.

6.1.2 Discussion

Development of the physics engine took a significant amount of time. It turned out to be a lot harder than expected to resolve collisions and rotations in a satisfactory way. Some desired features, like the ability to stand on moving platforms, were never implemented. It is hard to say for sure how much time it would have cost to use an existing physics engine, but it would likely have reduced the development time and allowed for more time to be placed on other things. In the end it was not completely wasted time, as knowing how the physics worked internally did help when designing movement mechanics, but in future projects an existing solution will likely be used.

6.2 Updating the Simulation

Analytical solutions for calculating the next step of the physics simulation are not usually possible [5, p. 692]. Therefore, the use of some sort of numerical approximation is necessary. One common method is the *explicit Euler* method, which works by assuming that the velocity and acceleration is constant during a timestep [5, pp. 692-693], see equations 6.1 and 6.2 for assumptions made.

$$pos + \Delta pos = pos + vel \cdot \Delta time \tag{6.1}$$

$$vel + \Delta vel = vel + acc \cdot \Delta time \tag{6.2}$$

6.2.1 Results

The numerical approximation used by Legend of Cube is the previously mentioned explicit Euler method. It was mainly chosen for its simplicity and ease of implementation.

```
// Example, not actually stored like this
struct PhysicalEntity {
    Vector3 position;
    Vector3 velocity;
    Vector3 acceleration;
}

// Very simplified implementation
void Update(List<PhysicalEntity> entities, float delta)
{
    foreach (var entity in entities)
    {
        entity.velocity += gravity * delta;
        entity.velocity += entity.acceleration * delta;
        entity.position += entity.velocity * delta;
    }
}
```

Listing 6.1: Example code showing how the simulation is updated using explicit Euler.

As one can conclude from Listing 6.1, the velocity and acceleration vectors are not modified by the basic simulation (though velocity may be changed during collisions). Therefore, the acceleration and velocity variables are often used as input for the physics system from other systems. For example, the movement system modifies both variables in order to perform jumps.

6.2.2 Discussion

The choice to use explicit Euler turned out well. Due to the simplicity of Legend of Cube's physics engine, no particular problems were encountered. If a more complex physics simulation had been used, a better approximation might have been more appropriate.

6.3 Bounding Volumes

An important part of a platform game is the ability to interact with the environment, for example jumping on platforms. In order to accomplish this and give the impression of solid geometry, the physics engine needs to be able to detect and resolve collisions. Collision detection is a subject that covers various algorithms and techniques used to determine if, when and where two objects collide [25, p. 1].

It is often too expensive to calculate intersections using the actual geometry of the game world [25, p. 75], instead *bounding volumes* are used. A bounding volume (henceforth abbreviated as BV) is a representation of a simple volume in space, e.g. a sphere or a box. In general, it is cheaper to calculate intersection tests on BVs than on more complex geometry [25, p. 75]. The idea is to fit the more complex geometry to BVs and then only compute the more complex intersection tests when necessary.

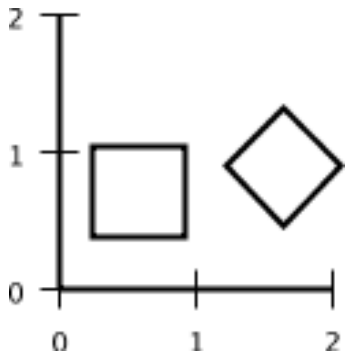
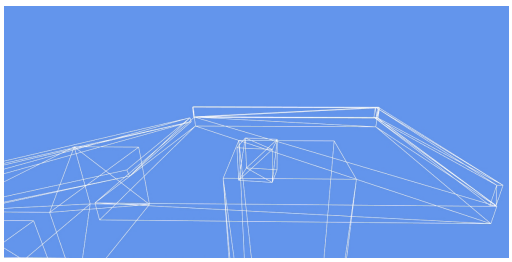


Figure 6.1: The left box is an AABB and must therefore be aligned with the world's coordinate system. The right box is an OBB and can therefore be freely oriented.

Two common types of BVs are *axis-aligned bounding boxes* and *oriented bounding boxes* (henceforth abbreviated as AABB and OBB respectively). The main difference between these BVs is that OBBs can be rotated arbitrarily in the world, while AABBs must be aligned with the world's coordinate system, which is illustrated by figure 6.1. Additionally, the intersection test between two OBBs is more expensive than the AABB equivalent [25, pp. 76-77].

6.3.1 Results

For simplicity's sake, only one BV is allowed per object and only OBBs are used. Since the player character is a cube, a single OBB is a perfect fit and more complex intersection testing algorithms are therefore not necessary. Additionally, the most important structures in the game, i.e. walls and platforms, can easily be represented by a single OBB. By using only one BV per object and sticking with only one type of BV, the software architectural complexity is reduced while still allowing for accurate player collisions. Figure 6.2 shows how OBBs are fit to the geometry in Legend of Cube.



(a) In-game screenshot showing OBB BVs.

(b) In-game screenshot showing the same scene using normal rendering.

Figure 6.2: Comparison showing how OBBs are fit to the geometry in Legend of Cube.

6.3.2 Discussion

Early during the development process, a choice between using AABBs or OBBs as the primary BV was made. This was a fundamental choice about the nature of the levels and gameplay in the game. Choosing AABB would mean forsaking rotations of both the player cube and the levels themselves. Similarly, structures like slopes would not be possible. This would not necessarily be a bad choice. A limiting choice like this could potentially make the whole design more clear and intuitive.

Ultimately, it was decided to only use OBBs in Legend of Cube, since they offer a larger amount of freedom in the shapes they can represent. One disadvantage of this approach is, as previously mentioned, that the intersection test between two OBBs is more expensive than the AABB equivalent. It was deemed that this was not a major problem, as there were more important optimizations available if needed.

6.4 Contact with Surfaces

In a platform game, the player can often perform different actions depending on whether the character is in contact with a surface or not. For example, if the character is on the ground a jump can be made, or if the character is sliding on wall, a wall jump could be performed. It is therefore crucial that these states can be accurately queried from gameplay systems. The player will not feel in control if a jump is not registered right before falling over the edge of a platform.

6.4.1 Results

The physics system in Legend of Cube determines if the player character is in contact with a surface by checking if a collision occurred. This poses a problem if the character is touching a surface but not actively moving towards it since no collisions will be detected. Likewise, if the velocity towards the surface is low enough it might not register as a collision and the contact state will be off for a single frame.

A couple of modifications were introduced to combat these problems. The first modification slightly increases the velocity into the surface every frame after the first collision. This decreases the chance of the physics system temporarily missing a collision due to the velocity being too low. The second modification attempts to displace the player character a small distance into the last known collision axis, to see if it is still in contact with the surface in question.

6.4.2 Discussion

The way contact with surfaces is handled in Legend of Cube could have been better. The modifications that were made to make it work are very specific, and it is difficult to know whether there are side effects yet to be discovered. To ensure that there are no such problems, the contact-detection would likely need to be rewritten

completely. This is probably one of the areas that would be improved by using an existing physics engine.

7

Graphics

The visual appearance of a game is vital for supplying the user with information of what is happening in the game world. The main idea behind the visual style of Legend of Cube was one of a gritty, dirty, industrial area with the cube being very high tech and alien looking, to contrast the surrounding environment. Prior to development, concept art was created to visualize this idea, as shown in Figure 7.1. The world is populated with exposed brick walls, piping, corroded metals and various construction materials whilst the cube is of a grey color with a blue light emitting from it. For creating a game with this visual appearance, it is considered throughout the development aspects that affect the visual experience. In the following sections, those aspects of the game's development are detailed.

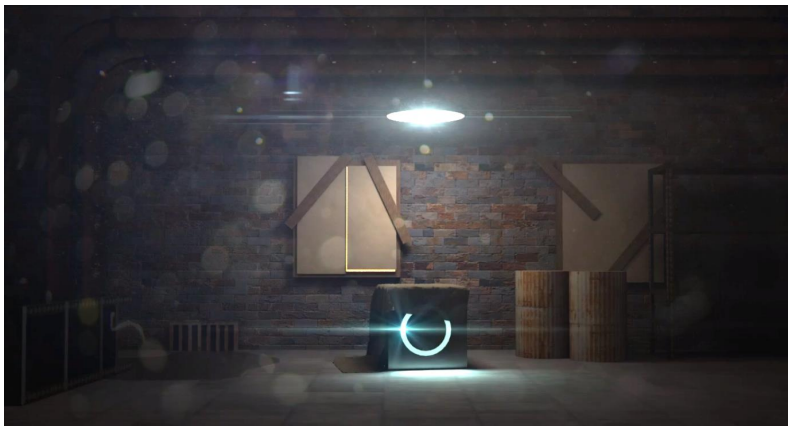


Figure 7.1: 3D rendered concept art showing the cube in an industrial environment.

7.1 3D Models

In 3D games, models are generally used as the main graphical components, whereas 2D games utilize forward facing images, or animations, called sprites. 3D models consist of points, *vertices*; and connecting lines, *edges*, which describe and define their shape. As computer hardware becomes more powerful, it is possible to have more vertices and edges, which thereby enables games to use more complex, and realistic, shapes.

7.1.1 Textures

When a 3D model is complete, it is often textured before being added to the game. Texturing involves applying a color or, more complexly, an image to define the surface of a given model. A model is not limited to a single texture. In order to create a more realistic look, different textures, or maps, are applied to the models. Each map contributes in its own way to the final surface, some define how glossy the material appears, and others the roughness of the surface.

7.1.2 Result

As discussed in section 6.3.1, Legend of Cube makes use of OBBs only, and in addition to that, each entity is restricted to one OBB and one 3D model. This puts some constraints on the game's 3D models since they cannot be of complicated shapes if the player is expected to collide with them.

Collisions with misaligned or ill-fitting OBBs cause the models to intersect – if the OBB is too small. If the OBBs are too large, they limit the models' minimum distance to each other – causing them to collide even though, visually, they do not appear close enough. Therefore, only models whose exterior resembles a cuboid can have a perfect fitting OBB and the relatively simplistic models in Legend of Cube is a direct result to that.

In Legend of Cube there is support for up to four maps on any given model – namely the diffuse-, normal- (see 7.4.1), specular- and emissive maps. See section 7.4 for what each map represents.

7.2 Camera

All games present a set of challenges for the camera system. These challenges includes how a player will interact with the camera, if at all, and how the camera will behave in edge cases, such as when there is little to no room for the camera to operate in, despite movement being required.

This section aims to present information on design decisions developers must make by exploring some of the different alternatives available. Then the camera in Legend of Cube is detailed, by first going through the specific camera requirements for the game, followed by how the camera then was implemented.

7.2.1 Camera Representation

One common type of virtual camera is one based on perspective projection, which simulates the function of real cameras. This is a type of camera that can be seen as being located at a fixed point in a virtual world with a particular orientation. This camera could be modelled in various ways, depending on the specific domain and purpose. One model of such a camera can be seen in Figure 7.2.

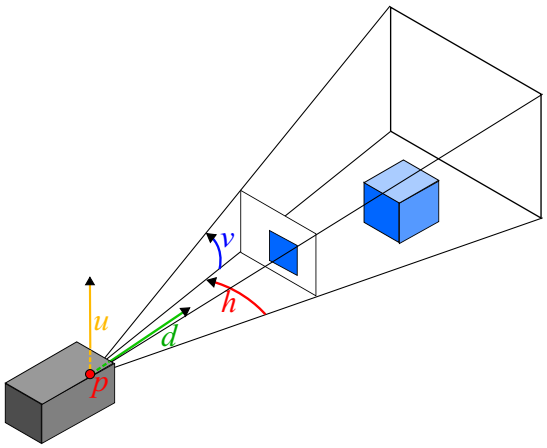


Figure 7.2: A representation of a virtual camera, with variables marked and labeled.

The box on the lower left in Figure 7.2 is where a physical camera would be, however it is not modelled as a three-dimensional object, but rather only as a point. The variables in the figure are defined as following:

- Position, $p \in \mathbb{R}^3$, relative to the center of the world.
- View direction, $d \in \mathbb{R}^3$, describes the orientation of the camera. This could additionally be referred to as *look-at* direction, as by Christie et al. [26].
- Up direction, $u \in \mathbb{R}^3$, which describes the roll of the camera, or rather what is upwards in the camera's perspective.
- Horizontal field of view, $h \in [0, \pi)$.
- Vertical field of view, $v \in [0, \pi)$.

7.2.2 Movement Mapping

In order for players to move the avatar in the game world, the directional input (up, down, left, right), must be translated into vectors in the world's coordinate system. Mapping movement controls can be done in many ways and tend to differ between genres. One way of orienting the movement is to align the "forward-vector" to the camera's forward direction – this way, the avatar moves away from the camera when players input a forward command on their control device [27]. Following this control scheme, a player who inputs "right" will subsequently cause the avatar to move to the right on screen.

In many racing games, you can often find another control scheme. Here, the controls are mapped to the vehicle itself in the way that "up" control the throttle and thereby moving the vehicle forward along its own axis, independent of the camera position or angle. Examples of games using this scheme are *Burnout Paradise* [28] and *Race Driver: Grid* [29].

7.2.3 Interactive Cameras

If developers wish to have a camera that the player can control, they must define what aspects of it the interactivity will be limited to, if at all, and to what degree. Christie and Oliver [30] state that a camera has seven degrees of freedom – three of which correspond to location in the three dimensions, three to orientation namely tilt, roll and pan, and last the seventh degree, which is the field of view degree. Most of the time, limiting what degrees are available for the player to control, yield better results than letting players assume full control of the camera. By doing so, the player can focus on gameplay, instead of camera angles.

It is common practice in third-person 3D games that players will have control over the camera's orientation, although sometimes to a very limited degree, but not its position – similar to a tripod. This way, players can examine the environment whilst the camera remains at a fixed point, and only moves if the player moves the avatar. Using this model, players will be less likely to lose track of the avatar when it exits the frame.

An interactive camera grants the players permission to explore the world further, but does in addition pose a set of problems of its own. Unlike a movie, developers can not guarantee that players will look at what they are intended to. This could in theory lead to confusion on the players' part, since they can potentially fail to notice important details.

There are ways to force the player to look wherever the developers need them to, one of which is using jump cuts or cut-scenes, and to that way take full control of the camera, which provides opportunity for greater cinematography [30]. Another way of displaying critical information is to limit the degrees of freedom available to the player. This way, developers avoid taking away all control from the player but limiting their vision, to e.g. a specific direction, in order to convey critical information. When making changes to the camera, developers must not make the transitions too abrupt. Christie and Oliver [30] rightly point out that changes to the camera's location or orientation must transition smoothly to avoid disorienting players.

7.2.4 Camera Behavior

In interactive applications, such as a game, events are not deterministic, but rather depend on the actions of the user. This rules out the possibility of defining the exact behavior of the camera in a scene, but rather there needs to be some algorithm, or artificial intelligence, that determines the camera's location in the scene over time.

In a typical 3D application with a virtual environment, such as a 3D game, there is often an underlying loop that continually updates a virtual world. The logic for updating the camera could conceptually be seen taking place toward the end in such a loop, as is shown in Listing 7.1. This is similar to the proposal, by Halper et al. [31], to consider a "camera module" as a part of a game pipeline, with five abstract steps, given in the order: *story*, *action*, *camera*, *lighting* followed by *render*.

```
void Loop()
{
    FetchInput();
    GameLogic();
    UpdatePhysics();
    UpdateCamera();
    RenderWorld();
}
```

Listing 7.1: A simplified game loop in which the relative location of camera updates can be seen.

In some games, there are cameras that, over time, smoothly catch up to the player. One such camera implementation is detailed by Rabin [27], which describes the camera in *Super Mario 64*, a platform game from 1996 [32]. The relation between the player and the camera in Super Mario 64 is described by Rabin as the camera being connected to the player character with a bungee cord, while in addition floating vertically in the air somewhat above the player.

7.2.5 Camera Requirements

For LoC, there are several high-level requirements that the camera must fulfil. Firstly, it must always adequately convey where the player is, relative to its surroundings. Another important aspect is that its behavior is predictable. This makes it possible for the player to learn, subconsciously, how the camera moves in different situations. This means that the player can focus on other aspects of the game that are actually meant to be challenging. Related to predictability is that the camera should act smoothly, as a continuous motion. There should be no noticeable stuttering or instant movement of the camera to a new location during regular gameplay.

Additionally, the camera needs to be interactive. This is so that the player can look around and plan their route through the level, which is considered essential for a platform game. However, the game should be playable without manual camera interaction as well. This presents a challenge for the camera management in the game, as well as a factor to take into account when designing the levels.

7.2.6 Result

The camera in Legend of Cube is specifically a third-person camera. The main idea behind the behavior of the camera is that it moves the shortest distance possible to get in a certain distance behind the player. This is similar to the idea behind the camera in the previously mentioned Super Mario 64, as well as a number of newer similar games, such as *Psychonauts* [33].

Internally, the camera is represented in a similar fashion to what is described in section 7.2.1. One difference though, in LoC, as well as in other games, is that the horizontal and vertical field of view (FOV) is dependent on one another. This is because both the screen width and height are given constraints, and to fill the available area while showing the scene proportionally, one must necessarily depend

on the other. For LoC, the decision to control the vertical FOV was made. The default vertical FOV is 70°, or $\pi/3$ radians, but it is user-selectable.

The mapping of movement input to character movement is relative to the camera's perspective. So that, giving the input to move *forward* on the controls will make the player character move straight away from the camera, and vice versa.

Although the camera will follow the player character, the user can still control it manually. Essentially the camera is in a different mode than previously described, while the user is controlling the camera, as well as a short time afterwards. In this mode, the camera's position relative to the player will *only* be changed by the user's input, and the camera will not automatically fall behind the player while moving.

As opposed to some third-person games, the camera is not controllable by mouse, only by keyboard buttons, when using mouse and keyboard. This is partly to make it clear to the user that the camera is not meant to require manual adjustment during normal gameplay. Using a gamepad, the normal stick used for moving the camera can be used.

When it is determined that a property of the camera should change, a smoothing method is sometimes used. It can be referred to as *critically damped* oscillation, e.g. by Haberman [34], and can be described as releasing a loaded spring, and not having the spring movement pass the equilibrium point (or resting point), only gradually getting closer over time. In the discrete logic of the game, it is essentially implemented as given by the relationship

$$x_{n+1} = a \cdot x_n + (1 - a) \cdot x_t$$

where x_n is the camera property undergoing change at time nth iteration of camera updates, x_t is the current desired resting value, and $a \in [0, 1]$ is a smoothing factor that determines the rate of change. This technique is, for example, used when making the camera rest at an angle slightly above the player character.

7.2.7 Discussion

We found that the camera implementation worked fairly well for our primary level. It is however likely that if we were to design more levels, new issues would arise. The camera is limited in several ways, and there are some obvious potential improvements. Mainly, there is no check for if the view of the player is occluded or if the camera is on the inside of other objects. This is commonly considered in other games and would be implemented if more time was available.

7.3 Real-time Rendering

For most 3D games, there need to be a way to translate the state of the game world to a two-dimensional image. For a physically realistic appearance, this conversion often involves simulating the phenomenon of photons moving in the world and

finding their way into a camera or an eye, which can be seen as a physical process that generates a two-dimensional image from the state of the world.

For a game performance is essential, as discussed in section 5.3. This rendering process is where performance likely is most important to be aware of. That which is practically possible in real-time applications, with consumer-grade hardware, is far from a theoretical, physically correct simulation of light phenomena. To achieve continuous conversion of a 3D world to pixel colors, at a high rate, there exists specialized hardware, a *graphics processing unit* (GPU), and a corresponding graphics pipeline [35, pp. 11-27]. This hardware is often designed for use with the *triangle rasterization* technique, which has found use in real-time 3D applications [35, pp. 11-27]. The general idea of triangle rasterization is to geometrically transform triangles to a space relative to the virtual camera and approximate the vector shape to a discrete set of pixels, which then are colored through the process of shading, described in section 7.4.

7.3.1 Result

For LoC, the rendering is similar to what is commonly used in games. That is, it uses triangle rasterization, accelerated by the GPU hardware. This allows for having large-scale and complex 3D environments. More precisely, rendering is done through DirectX 9, which is what XNA is based on [36].

7.3.2 Discussion

There are limitations of XNA that were noticed during the development process. Microsoft has not released an update to the framework since 2011 [37]. There would likely have been advantages with using a more recent, up to date, foundation for the game's development. This is especially true in relation to graphics. However, we still feel that XNA is a well-developed framework that strikes a reasonable balance between usability and performance. We consider it a good fit, considering the scope of the project.

7.4 Shading

For each pixel rendered, there is a point on an object in the scene it corresponds to. As each object is represented by triangles, the point is somewhere on this triangle. This triangle is colored by using a type of *shader program*, which are programs running on the GPU. Particularly *fragment shaders* or *pixel shaders* are involved in this process, as described by Akenine-Möller et al. [35, pp. 42-44]. Besides a pixel shader, what is commonly needed as well is a *vertex shader*. This is, described in a simplified way, a shader that transforms the triangles of the graphical model, stored relative to the model center, to their final position, relative to the camera perspective.

There are different types of light sources that could be used in a virtual environment. Two common ones used are *directional* and *point* light sources. A directional light source can be described as casting light from infinitely far away in some direction, so

that light rays would be parallel [35, pp. 217-218]. This can be used to simulate the sun's light on earth. A point light source on the other hand is one that is infinitely small and cast light in all directions.

Light simulation in 3D games is commonly based on the *Phong reflection model*, presented by Phong [38]. It is a model for simulating shading in a scene caused by a light source. It can briefly be described as dividing the shading into contribution of the following three components [39]:

- *Ambient*: Light that reaches everywhere in the scene. This is an approximation of the fact that light will reflect off of surfaces and light up parts of the scene that are not directly illuminated.
- *Diffuse*: Light that hits a surface and scatters in all directions up from the surface. The diffuse part would be greater for objects with a rough surface.
- *Specular*: Light that hits a surface and reflects off along a reflection vector. As opposed to diffuse, the specular highlight is greater for objects with a smooth surface.

In addition to this, *emissive* light can be added, which, as ambient, adds to the color of materials, independent of a light source. It can additionally be referred to as *self-illumination* and is used for objects that should glow on their own, but still are not implemented as a full light source, which in addition would light up the nearby environment. For an illustration of the four mentioned types of light, see Figure 7.4.

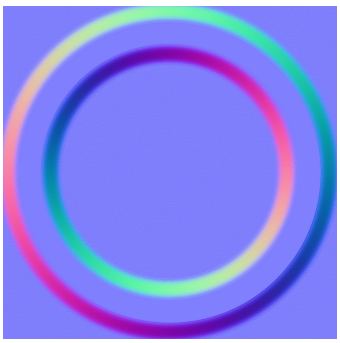
A modified technique based on the Phong reflection model was proposed by Blinn [40], and is referred to as the *Blinn–Phong reflection model*. The difference relates to how the specular component is determined.

7.4.1 Normal Mapping

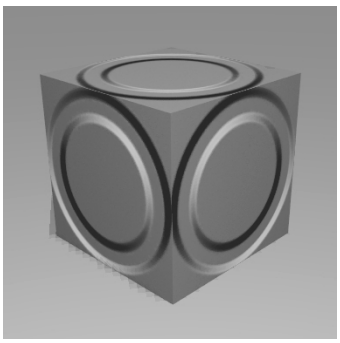
Normal mapping can be considered an implementation of a more general technique called *bump mapping*, according to Akenine-Möller et al. [35, pp. 183-199]. The general idea is to simulate more detailed 3D models, without actually increasing the amount of polygons used, by letting the per-pixel lighting depend on a texture that contains detailed information about the surface shape. As the name suggests, smooth *bumps* on a surface could for example be simulated using this method.

What differentiates normal mapping is related to how the information is stored in the texture [35, pp. 183-199]. One simple method is to store the height of different parts of the texture in a black and white *heightfield*, where the brightness relays the relative height at different points of a surface. For optimization reasons, using modern GPUs, normal mapping is an alternative preferred format. Here, the *red*, *green*, and *blue* channels of a texture describe the normal pointing up from the surface. Commonly, the normal is stored in *tangent space*, which, with some simplification, means that the normal is stored relative to the surface polygon it appears on. For an example of normal mapping, see Figure 7.3. There, it can be

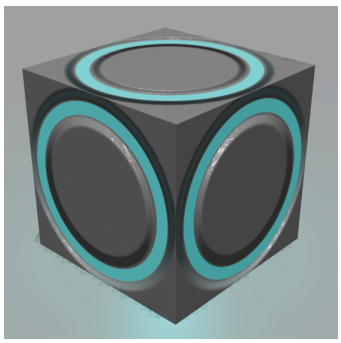
seen that the technique is less effective when viewing a surface from a sharp angle, like the top of the cube.



(a) Normal map for an indented ring.



(b) Shaded cube model (only 8 vertices) with a normal map applied.



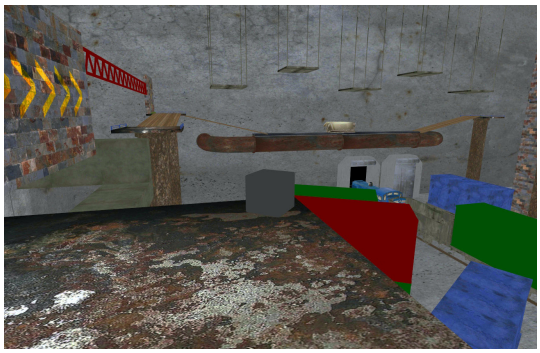
(c) Final rendering using additional textures.

Figure 7.3: Example of normal mapping, showing a cube being lit from behind the viewer. The model is one used early in development.

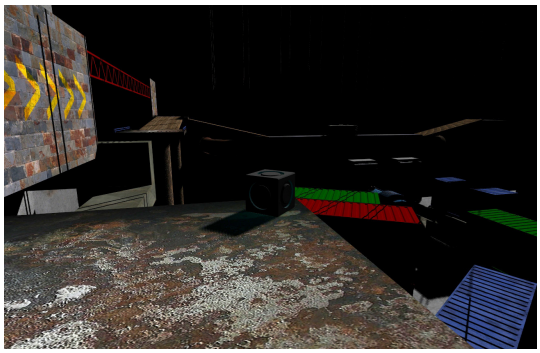
7.4.2 Result

The XNA framework provides *XNA effects*, which corresponds to one or more shader programs. An option was to use one of the included effects for LoC. However, it was found to be too limiting for what was needed in LoC. For example, it would not be possible to have shadows with the included effects. For that reason, a custom XNA effect was created, which is then used for rendering most game objects. Making the XNA effect involved writing a custom vertex shader, as well as pixel shader.

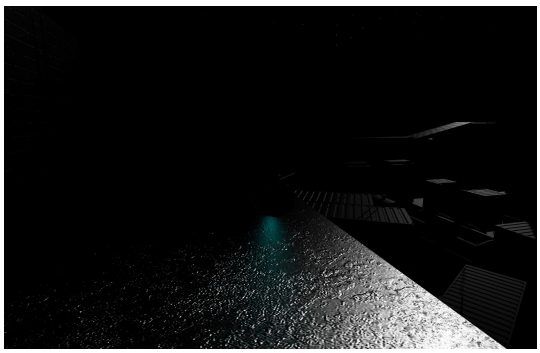
The intended look for this game was to have the scene look realistically shaded, rather than a more abstract, non-photorealistic, appearance. For achieving that, the Blinn-Phong lighting model is used, by implementing it in a pixel shader. Figure 7.4 illustrates the different parts that are used to achieve the final result.



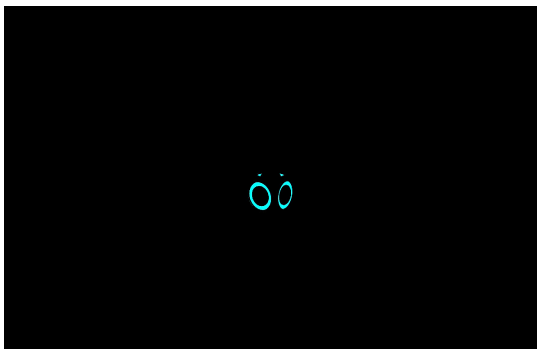
(a) Ambient component



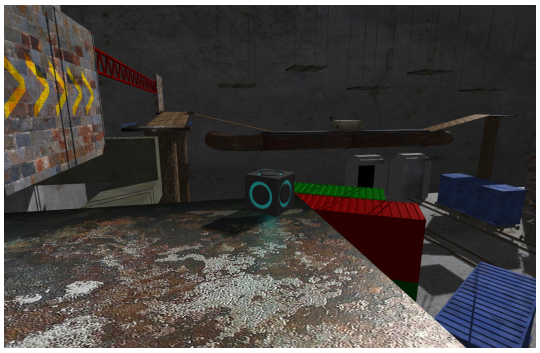
(b) Diffuse component



(c) Specular component



(d) Emissive component



(e) Resulting image

Figure 7.4: Scene in LoC where different light components are shown separately.

The scene seen in Figure 7.4, and all levels in our game, are lit by two different light sources. The first one is a directional light source, covering the entire scene. The second one is a point light source that is located in the center of the player character. This is to enhance the effect of the cube itself giving off light from each side, in addition to the emissive effect seen in 7.4d. Due to being a light source, a highlight on the platform can be seen in 7.4c.

In the game, to improve the appearance of some objects, the normal mapping technique is used. For an example, see Figure 7.5. In it, the normal map enhances the wood pattern, as well as makes the separation between planks more apparent.

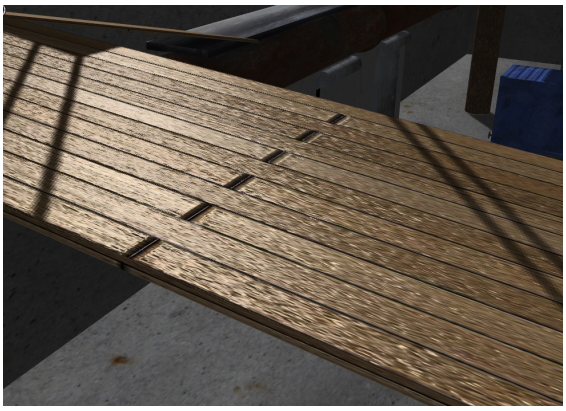


Figure 7.5: In-game screenshot of wood planks seen in Level 1.

7.4.3 Discussion

The visual appearance of the game is considered more than adequate for the gameplay-related goals of the game, namely having good control of the character. However, presenting the world in a clear way is vital for having the player be in control, since information about the physical surroundings affect how the player can plan their actions. In addition, we believe that improved visual appearance could significantly add to a user's overall impression of the game.

7.5 Shadows

As is described by Akenine-Möller et al. [35, pp. 333-373], shadows are an essential part of a realistically rendered scene. Using the triangle rasterization process, additional techniques need to be used for having shadows in rendered scenes.

Different techniques, with different advantages and disadvantages, have been developed for simulating shadows when using triangle rasterization. One technique used in some games is a method called *shadow mapping*. As described by Akenine-Möller et al. [35, pp. 333-373], the idea is to render the scene from the lights perspective and store the depth information, which will later be used for determining if other positions in the scene are located behind what can be seen from the lights perspective. This method utilizes the existing hardware that is normally used when rendering a scene, which contributes to the efficiency of the technique.

7.5.1 Smooth Shadows

Using the simplest implantation of shadow mapping, there is a problem of shadows showing *aliasing artifacts*, which can be described as the shadows looking very blocky, or pixelated. Presented by Reeves et al. [41] is a technique called *percentage-closer filtering*, abbreviated PCF, which attempts to solve some of these problems. In addition to shadows not appearing as blocky, PCF can be made to give shadows soft edges, which often is a desired effect.

An essential idea with PCF is that a point can be partially lit. When checking if a point is in shadow, instead of looking only at the exact location corresponding location in the shadow map, the shadow map is sampled at nearby texture coordinates as well. Then, the result is determined by averaging the result of the samples, which gives a result, commonly in $[0 - 1]$, that determine what *percentage* of the point is lit. This averaging will cause the smooth transitions from fully lit, to shadowed, which are desired.

7.5.2 Cascaded Shadow Maps

One limitation of the shadow mapping technique is that the textures on which the shadow maps are stored are of a limited size and resolution. A texture can be made to cover a larger area with lower precision, or a smaller area with higher precision. One observation that has been made is that greater precision will have the greatest improvement on the final image if it is applied closer to the camera. From this comes the idea of using multiple shadow maps that cover different parts of the scene, and for each position using the most detailed one available [35, pp. 333-373]. This is called *cascaded shadow maps*. It has been detailed by Dimitrov [42], and has been used in commercial games [35, pp. 333-373].

7.5.3 Result

As previously described, there are two light sources in the scene. Specifically, there is one directional, and one point light. For the directional light source, it was considered essential to have shadows, to give realism to the scene. The light point light source however was considered enough, so that it was acceptable to exclude the shadows it would cause. What is detailed hereon therefore only applies to shadows from the directional light source.

In Legend of Cube, the technique used for shadows is shadow mapping, with PCF filtering, and a simple version of cascading shadow maps. For example of shadows in the game, see 7.6, or the previously shown 7.4e and 7.5. The resolution of the shadow maps are fixed at 2048 pixels, both width and height. The PCF implementation is done explicitly in the pixel shader code, rather than using hardware accelerated PCF. This is due to DirectX 9 limitations.

For the levels in LoC, it was decided that more than one shadow map was needed for both covering a large area and to give enough detail close to the player character. This led to implementing a version of cascading shadow maps. Given the limitations of the game, it was decided to have only two differently sized shadow maps. One of them is set to be substantially larger than the other one. Both of them are of a constant size relative to the world. For determining what area each would cover, a very simple method was used. This is to center both shadow maps on the camera's position, rather than trying to optimally fit the shadow maps in the camera's frustum.

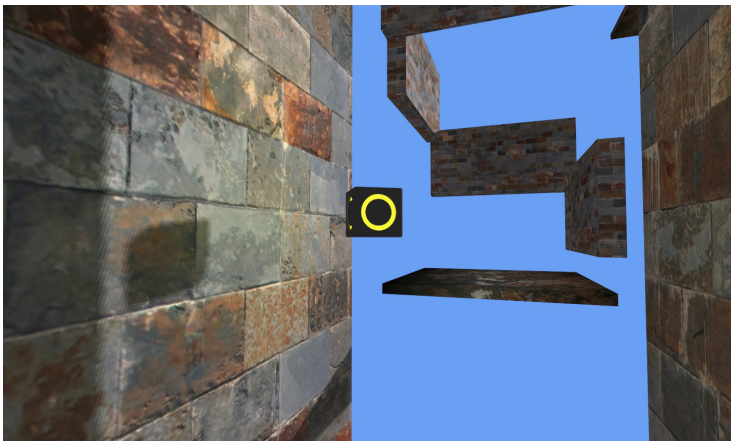


Figure 7.6: In-game screenshot of the player cube in the middle of a jump. A shadow of the cube can be seen on the left wall.

7.5.4 Discussion

In a game like LoC, shadows could not only be used for giving a visually convincing image. They can additionally convey the relative location and size of objects. For example, while the player cube is located in the air, if the shadow of the cube is projected onto another visible object, it will hint at their position in relation to each other. For that reason, we think the shadows can be useful from a gameplay perspective. For example, the shadow in Figure 7.6 could have this effect.

8

Results

A large amount of effort was dedicated to the platforming gameplay during development. The jumping and movement mechanics were evaluated extensively through playtesting with both developers and independent users – evaluating control and general playability. Based on feedback, both mechanics and the level, in which the tests were conducted, were changed to improve the user experience. These changes were often done by changing parameters within the physics engine, wherein everything from gravitation, to collisions, is handled.

The physics engine in Legend of Cube was created primarily for exploring different jump mechanics, and thus became somewhat limited. The physics engine worked fine, given the scope of the project, despite some issues with contact and collision detection. The limits of the physics engine, in conjunction with the software architecture, put some constraints on the 3D models of the game. If the player is to collide with another entity, that entity needs to assume the shape of a cuboid.

The third-person camera implemented for Legend of Cube moves the shortest distance possible in order to get behind the player, if it is not manually operated. Manual operation enables players to explore and see the way ahead when making sharp turns. What the camera shows is transformed to an image on screen by the rendering system, utilizing the GPU and a shading model commonly used in real-time applications. The process includes determining what parts of the scene are shadowed.

8.1 User Tests

As was described in section 2.3, user tests were performed in order to evaluate the gameplay mechanics. After playing between five and ten minutes, three participants had managed to complete the level while the rest had declared that the game was too hard and gave up. The participants were then interviewed about their experiences with the game. Interviews were performed since it has been shown to be a good way to explore issues users might have, e.g. by Preece et al. [43]. The participants were asked to rate their experiences in regard to the four aspects of gameplay mentioned above. Consistency, freedom and predictability all scored very high across all tests but intuition received a considerably lower mark, especially from participants using a keyboard. Upon further questioning, it was revealed that the reason for this was that the camera is controlled by the arrow keys and not the mouse, which is the status quo in modern PC games. Another reason was that some participants wished to invert the camera controls, but did not ask or check if such

an option was available. Participants who gave low intuition scores complained that it was sometimes difficult to know where to go next, one participant commented "Hard to see where the finish line was".

9
Discussion

The result of the user tests seem to show the goal of making the player feel in control has, at least to some extent, been reached. However, it also shows that the level was too hard for new players since only 25% of the test group could complete it. There are reasons to believe that there are other factors then just the level being too hard that prevented the testers from completing the level. Since many testers gave intuition a low score, it can be reasoned that this played into why many could not finish the level. Users using a keyboard often gave lower intuition score because they did not like the controls; this could be because the developers intended the gamepad to be the target controller. Another reason that users gave a lower intuition score was that they thought it was hard to see the finish line. This is a flaw in the level, since there is no established way in the game of how the goal should look, like a finish line in a racing game. Nevertheless, the developers believe that in future work an introduction level could be designed and implemented to introduce new players to the game and its different mechanics in a more controlled way, and that it would rectify most of the problems as to why many could not complete the level.

For a long while before the basic functionality of the physics engine was implemented, we feared that the resulting product would end up being a very basic prototype without any interesting platforming mechanics. However, once the most needed features of the physics engine were in place, such as collision detection and response, things started to come together. Looking back, it is very satisfying to see how far the game has come, from literally nothing, to being able to slide on walls and perform wall jumps.

For us developers, what we consider the primary success of this game to be how well it turned out as a whole. The movement mechanics are not perfect, but they fulfill our goal of having the player feel in control, which is supported by our user tests. The wall sliding and jumping mechanics in general are particularly satisfying to use. The visuals clearly communicate how the player can traverse through the level, while looking quite nice most of the time. The level design itself has multiple paths to the goal allowing for more replay value and speedrunning potential. While some more work could have been spent on making the camera behave perfectly, it still shows what is needed most of the time.

Looking back, one of the things we might have reconsidered is the use of the XNA framework. In general, its use seems to have been deprecated in favor of complete game engines, like *Unity* and *Unreal Engine*. Using one of those engines and an existing physics engine might have made it a bit harder to implement the

exact platforming mechanics wanted, but would probably have speed up overall development time and allowed us to focus more on the gameplay itself.

With a few more months of work, it is very possible that Legend of Cube could be turned into a product that consumers may want to play. As it stands now, the lack of levels is primarily what keeps the game from being complete. The data-driven aspects of the engine could be expanded to allow for externally defined levels, it might even be possible to create a level editor. Besides that, there are a few inconsistencies in the platforming mechanics and the camera that could be fine-tuned and improved to facilitate the players feeling of in control.

10
Conclusion

This thesis details several aspects of the development of a 3D platformer, with special focus on making the player feel in control while playing. The game Legend of Cube, developed by the authors, features one complete level, which highlights all of the game mechanics designed. It runs on a custom game engine featuring a physics engine, designed specifically to make it easier to explore potential game mechanics.

The focus of making the player feel in control has been reached through researching mainly game mechanics and level design, but the effects of graphics, camera and physics on this subject has also been investigated. Through user tests, it has been concluded that the goal of good player control has been, to some extent, achieved; however, the process of achieving perfect controls is arguably an endless journey.

The content of this thesis could be used to give insight about problem areas for anyone who intends to develop a 3D platformer. Arguably, the speed and accuracy of 2D platformers have not yet been matched in 3D. We feel that until this has been accomplished, there is still work to be done trying to improve platform mechanics in 3D. For that reason, the authors think that more studies on the topic of player control in 3D platformers are needed.

Bibliography

[1] L. Parker, "The rise of the indie developer", 2011. [Online]. Available: http://www.gamespot.com/articles/the-rise-of-the-indie-developer/1100-6298425/ (retrieved 2015-05-18).

[2] Team Meat, *Super meat boy*, 2010. [Online]. Available: http://supermeatboy.com/ (retrieved 2015-05-17).

[3] *Xna game studio 4.0 refresh*, Electronic Article. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb200104.aspx (retrieved 2015-05-11).

[4] MonoGame Team, *Monogame*. [Online]. Available: http://www.monogame.net/ (retrieved 2015-05-17).

[5] J. Gregory, *Game Engine Architecture, Second Edition*. CRC Press, 2014, ISBN: 9781466560062.

[6] *Xbox 360 controller for windows*. [Online]. Available: http://www.microsoft.com/hardware/en-us/p/xbox-360-controller-for-windows/52A-00004 (retrieved 2015-05-18).

[7] SpeedRunsLive, *Frequently asked questions - speedrunslive*. [Online]. Available: http://www.speedrunslive.com/faq/ (retrieved 2015-05-18).

[8] Nintendo, *Super mario 64*, 1996. [Online]. Available: http://en.wikipedia.org/wiki/Super_Mario_64 (retrieved 2015-05-17).

[9] Nintendo, *Super mario sunshine*, 2002. [Online]. Available: http://en.wikipedia.org/wiki/Super_Mario_Sunshine (retrieved 2015-05-17).

[10] Tinybuild, *Speedrunners*, 2013. [Online]. Available: http://tinybuild.com/speedrunners (retrieved 2015-05-17).

[11] T. Refenes, Team Meat. [Online]. Available: http://meyermike.com/wp/?p=160&cpage=1#comment-68 (retrieved 2015-05-17).

[12] *Super meat boy: customer reviews (steam store)*. [Online]. Available: http://store.steampowered.com/app/40800/#app_reviews_hash (retrieved 2015-05-17).

[13] Sourceruns, *Bunnyhopping*, 2015. [Online]. Available: http://wiki.sourceruns.org/wiki/Bunnyhopping#GoldSrc (retrieved 2015-05-16).

[14] C. Bleszinski and E. Games, "The art and science of level design", in *2000 Game Developers Conference Proceedings*, 2000.

[15] B. Bates, *Game Design (2nd Ed.)* Thomson Course Technology, 2004, ISBN: 1592004938.

[16] Dice, *Mirror's edge*, 2009. [Online]. Available: http://www.mirrorsedge.com/.

[17] M. Brandse and K. Tomimatsu, "Empirical review of challenge design in video game design", English, in *HCI International 2013 - Posters' Extended Abstracts*, ser. Communications in Computer and Information Science, C. Stephanidis, Ed., vol. 373, Springer Berlin Heidelberg, 2013, pp. 398–406, ISBN: 978-3-642-39472-0. DOI: 10.1007/978-3-642-39473-7_80. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39473-7_80.

[18] C. Fabricatore, "Gameplay and game mechanics: A key to quality in videogames", 2007.

[19] Number None, Inc., *Braid*, 2008. [Online]. Available: http://braid-game.com/.

[20] Valve Corporation, *Portal*, 2007. [Online]. Available: http://orange.half-life2.com/portal.html.

[21] Mike Bithell, *Thomas was alone*, 2012. [Online]. Available: http://www.mikebithellgames.com/thomaswasalone/.

[22] e. a. Ahmed BinSubaih, "A survey of 'game' portability", [Online]. Available: http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS0705.pdf (retrieved 2015-05-17).

[23] M. Acton, *Data-oriented design and c++*, A presentation by Mike Acton given at CppCon 2014, 2014. [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc (retrieved 2015-05-15).

[24] Microsoft, *Fundamentals of garbage collection*. [Online]. Available: https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx (retrieved 2015-05-15).

[25] C. Ericson, *Real-Time Collision Detection*. Elsevier Science, 2004, ISBN: 9780080474144.

[26] M. Christie, P. Olivier, and J.-M. Normand, "Camera control in computer graphics", in *Computer Graphics Forum*, Wiley Online Library, vol. 27, 2008, pp. 2197–2218.

[27] S. Rabin, "Classic super mario 64 third-person control and animation", in *Game Programming Gems 2*, M. DeLoura, Ed., Charles River Media, Oct. 2001, pp. 425–432.

[28] Criterion Games, *Burnout paradise*, 2008. [Online]. Available: https://www.mobygames.com/game/burnout-paradise (retrieved 2015-05-18).

[29] Codemasters, *Race driver: grid*, 2008. [Online]. Available: https://www.mobygames.com/game/grid (retrieved 2015-05-17).

[30] M. Christie and P. Olivier, "Camera control in computer graphics: Models, techniques and applications", in *ACM SIGGRAPH ASIA 2009 Courses*, ser. SIGGRAPH ASIA '09, Yokohama, Japan: ACM, 2009, 3:1–3:197. DOI: 10.1145/1665817.1665820. [Online]. Available: http://doi.acm.org/10.1145/1665817.1665820.

[31] N. Halper, R. Helbing, and T. Strothotte, "A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence", in *Computer Graphics Forum*, Wiley Online Library, vol. 20, 2001, pp. 174–183.

[32] Moby Games, *Super mario 64*. [Online]. Available: https://www.mobygames.com/game/n64/super-mario-64 (retrieved 2015-05-18).

[33] Double Fine Productions, *Psychonauts*. [Online]. Available: http://www.psychonauts.com/ (retrieved 2015-05-18).

[34] R. Haberman, *Mathematical models: Mechanical vibrations, population dynamics, and traffic flow: An introduction to applied mathematics*. 1998, vol. Classics in applied mathematics, ISBN: 0898714087.

[35] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Third Edition*. Taylor & Francis, 2008, ISBN: 9781568814247.

[36] Microsoft, *The XNA rendering pipeline*. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd904179 (retrieved 2015-05-14).

[37] Microsoft, *Microsoft XNA game studio 4.0 refresh*, 2011. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=27599 (retrieved 2015-05-14).

[38] B. T. Phong, "Illumination for computer generated pictures", *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, 1975.

[39] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics", *ACM Trans. Graph.*, vol. 1, no. 1, pp. 7–24, Jan. 1982, ISSN: 0730-0301. DOI: 10.1145/357290.357293. [Online]. Available: http://doi.acm.org/10.1145/357290.357293.

[40] J. F. Blinn, "Models of light reflection for computer synthesized pictures", *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, pp. 192–198, Jul. 1977, ISSN: 0097-8930. DOI: 10.1145/965141.563893.

[41] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering antialiased shadows with depth maps", *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 283–291,

Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37435. [Online]. Available: http://doi.acm.org/10.1145/37402.37435.

[42] R. Dimitrov, "Cascaded shadow maps", *Developer Documentation, NVIDIA Corp*, 2007.

[43] J. Preece, Y. Rogers, and H. Sharp, *Beyond Interaction Design: Beyond Human-Computer Interaction*. New York, NY, USA: John Wiley & Sons, Inc., 2001, ISBN: 0471402494.