Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Controlling a Physics Based Mech Simulation in Room-Scale Virtual Reality

Andreas Leitner

Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Steuerung einer physik basierten Mech Simulation in Room-Scale Virtual Reality

Controlling a Physics Based Mech Simulation in Room-Scale Virtual Reality

Author: Andreas Leitner

Supervisor: Prof. Gudrun Klinker, Ph.D.

Advisor: M.Sc. Sandro Weber

Submission: September 15, 2017

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2017

(Andreas Leitner)

Acknowledgments

I would like to thank Prof. Gudrun Klinker and Sandro Weber for allowing me to write this thesis about the topic I presented them.

I would also like to say that I am very grateful that the Informatics: Games Engineering course of studies exists, and that it offers its students the possibility to try out new technologies like virtual reality and augmented reality.

Abstract

The goal of this thesis is to implement an immersive physics based mech simulation that is controlled in Virtual Reality. The user stands in the cockpit and can control the mech and its hands. To increase the realism and improve the interaction with the environment every part of the mech is physics based. This means that physical forces have to be used to keep the mech parts at the desired rotation.

The thesis first explains how the information we get from the user is utilized to control the simulation. This is more complex than most in mouse and keyboard games because we can track the position and rotation of both hands, the head, and additionally have six buttons and two touchpads on the controllers. After that the techniques that are used to control the physics simulation are explained in detail. Lastly the design process to the final mech models is described.

Controlling a Physics Based Mech Simulation in Room-Scale Virtual Reality

Chapter 1

Introduction

This chapter explains the reasons for the whole idea in more detail. The tools used to create the project are also described.

1.1 Motivation

The biggest challenge for every Virtual Reality application is to achieve a good and suitable way of moving within the virtual world. Most games and experiences choose to use the teleportation locomotion style because it is relatively easy to implement and is basically free of motion sickness. It can sometimes be hard to integrate it into action based gameplay and many users do not particularly like it. An other often used method is artificial locomotion, were the player movement is controlled the Vive touchpad or an other controller. This method is known for causing motion sickness and is unplayable for some users.

Then there is the option of having the player inside a cockpit and giving him a way of moving the vehicle he is sitting in. This has huge advantages because it can be very immersive if done right, and is less likely to make you motion sick because you always have parts of the cockpit in your peripheral vision. This also offers the developer many possibilities for movement. Cockpit games often do not fully utilize Virtual Reality because they mostly use VR as a more immersive way of displaying the game with the possibility to look around.

The original incentive for this project was to create a cockpit game that still utilizes the tracking of head and hands in the whole room. The chosen game mechanic that suits this very well is to let the player control a mech[1] and its arms. The main goal of this idea is to make the player feel like he is directly controlling the mech. In order to achieve a realistic arm control and interaction with the environment, the mech parts are fully based on physics simulation.

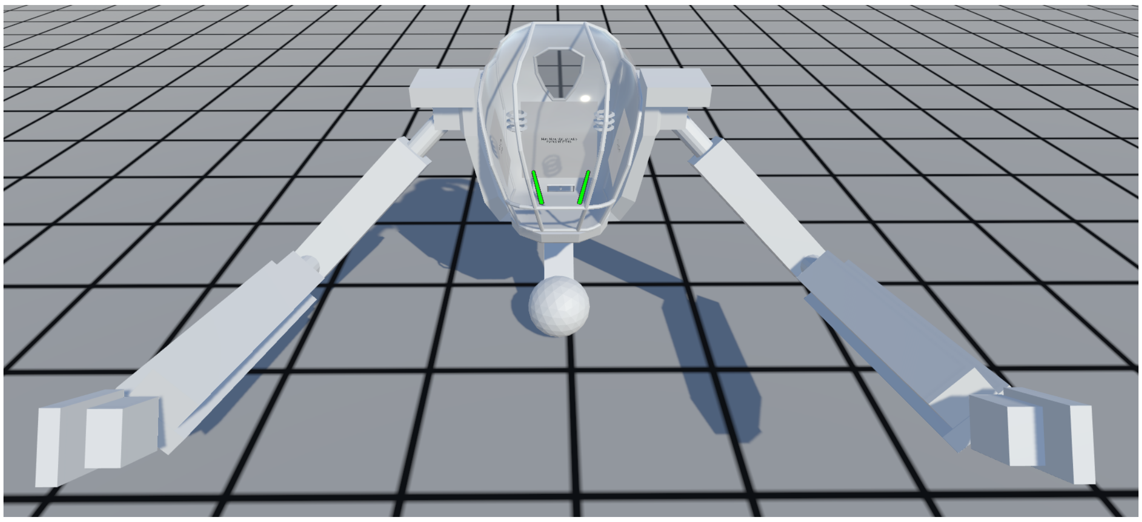


Figure 1.1: The mech the thesis talks about. The arms consist of a hand, a forearm and an upper arm. For more about the mech design see Chapter 4.

This thesis will explain the process behind the development of this project.

1.2 HTC Vive

The HTC Vive was chosen as a VR device for this project because it has extremely accurate tracking within a big play area. With their Lighthouse technique it can also track the controller no matter the orientation of the player. HTC have also recently introduced extra Vive Trackers that can be very useful to track the feet or back orientation and position.

Because this project uses Steam VR it is playable with all VR platforms that support Steam VR.

[1]A mech is a robot with humanoid appearance, which is often controlled by a human.

1.3 Unity

Unity is used as a game engine for this project, mainly because I am already familiar with it, and it is very good at rapid prototyping. The Virtual Reality integration is also very easy with the official "SteamVR Plugin"[1] and allows to test your game without a VR device connected. This is very convenient when testing or debugging because it is not necessary to take the HMD (Head-mounted display) on and off every time you want to try whether something works like intended.

Chapter 2

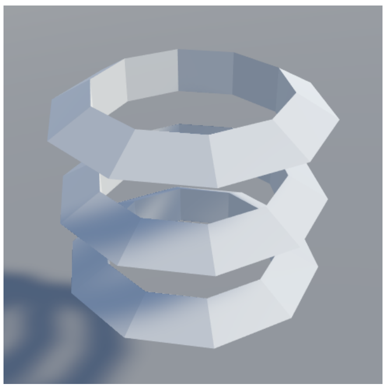
User Interaction

This chapter will explain how the user can interact with the simulation and how it was implemented. First it will explain the two modes the user can be in. Then it will describe how the head and hand position of the player are used in the different modes.

2.1 Game modes

The game has two separate modes. In the cockpit mode the player can freely walk inside the cockpit and can also interact with the user interface. Since the cockpit is relatively small[2] there is no need for an extra locomotion system because the user is already be able to reach everything he is supposed to reach.

Figure 2.1: The handcontroller that can be picked up to control the mech arms.



The second mode is the mech mode. In this mode the user can control the mech position and rotation and also acquires full control over the mech arms and hands.

The transition between the modes is handled with handcontrollers. They are objects in the cockpit the user can pick up with each hand, and enable the control of the respective mech hand.

[2]The ground of the cockpit is about 1.3 m wide and 1 m long

2.2 Head Position

To obtain the head position of the user it is mostly sufficient to use the position of the HMD. For applications that need a very accurate position to simulate body parts, it is important to understand that the position of the center of the head is about 14 cm behind the HMD since it is positioned on the face of the user. With this correction the head position we use stays about the same when the user rotate the head sidewards or upwards.

In the cockpit mode the user can freely move around in the cockpit and interact with the user interface. Since the user is also supposed to be able to interact with the back of the cockpit the mech does not rotate during this, otherwise the user would only be able to see the front of the cockpit. The mech arms are not controlled while the user is in this mode.

In the mech mode the position of the head is always fixed in the middle of the cockpit. The reason for this is to make him feel like he is the mech, and he is only controlling the mech, so separating the different movements is helpful. The movement in the play area is added to mech movement, so when the user takes a step to the right, the mech then also moves to the right. Since the user is limited by his play area this can not be used as a main movement method. In addition to this the touchpad of the Vive controller is used to additionally control the speed of the mech. It was also tested to lean the mech in the direction the user moves away from the center. This idea was quickly discarded after a test run because it was a very nauseous VR experience, since it felt like falling in the direction one moves.

The main challenge of the fixed head position is to keep the user inside his play area and also to encourage him to return to the middle of it. This is necessary because the player has no idea where he is in the real world since his virtual position is fixed to the middle of the cockpit. The first approach was to mark the center with a colorful circle on the ground, with an line to the user. But since this was on ground height the user would only see it when he was directly looking down, so it was not making the user aware of his surrounding.

The current solution is to have a miniature of the player in his play area boundary as a HUD (Head-up display) element. It is positioned on the glass front of the cockpit, so it is nearly always in the view of the player. The miniature consists of the body, head, and arms of the user, and is surrounded by the play area boundary. For the arms the same Inverse Kinematics is used as for the mech. There is also an arrow that points from the body of the miniature to the center of the play area. The boundary is turning red when the user comes too close to the border to furthermore warn the player. Since the boundary is probably not always in the sight, the hand-controller model that is around the Vive controller when the player controls the mech, also turn red. This works really well to keep the user aware of his position in the real world, and is also far more effective than the standard grid that Steam VR paints when the user comes to close to the boundary.

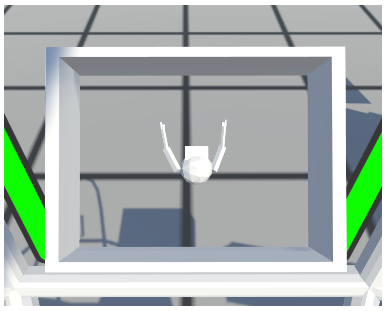


Figure 2.2: The miniature with a 2 m by 3 m play area.

2.3 Hand Position

In the cockpit mode the hand can be directly moved in the cockpit and can be used to interact with the user interface. The transition to the cockpit mode is triggered by picking up both of the handcontrollers (see Figure 2.1). When picking up only one controller, the user can control one hand of the mech. This was introduced because it feels great to first pick up both mech hands and not press a button to suddenly control the hands. It could also be used to quickly interact with the user interface.

The main feature of this simulation is the control of the mech arms, so it is very important that they feel good. The overall goal is to give the user the feeling that the mech arms are an extension of his own arms. The mech hand position is calculated as follows. The vector from the users head to each of his hands is calculated and scaled by a fix value such that fits to the length of the mech arm. This vector is now added to the position of the cockpit. In order to make it fit the proportions of the mech the resulting position is offset in the forward and upwards direction. The exact values were chosen such that the user can utilize the distance of the mech arm. This means that if the user stretches his arm as far forward as possible, then the mech is supposed to do a similar movement. The same should happen in all directions. The reason for this is to give the user the feeling of being able to utilize the mech arms in full potential. The forward offset is necessary because the front of the cockpit is about one meter in front of the shoulders and when the user has his hands in front of his chest the mech hand is supposed to be close to the front of the cockpit.

The hand orientation is set to the controller orientation rotated by a factor such that the hand looks forward when the user holds the Vive controller comfortable in his hand. To ensure that the hand can always rotate like the user wants there is no collision between the mech hand and the forearm. Objects like a sword hilt could also collide with parts of the arm so the collision between this two objects is deactivated to assure free hand rotation.

2.4 Inverse Kinematics

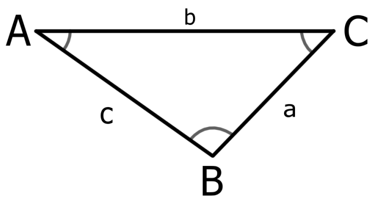
In order to control the mech arms an inverse kinematics algorithm is used, because we only know the position the hand should have. Since the cockpit orientation is directly controlled we have the following informations:

- The shoulder position,
- the length of the forearm and upper arm,
- and the position the hand is supposed to be.

What we need is the rotation of the forearm and upper arm such that the hand is at the target position.

Since this specific problem only has two unknown objects we do not have to use complex inverse kinematics algorithms that support more than two unknowns. To solve this problem the distance between the shoulder and the hand target is used to calculate the angle the arm has to be bend in order to have the right distance to the target. This is a basic trigonometric problem illustrated in Figure 2.3. Since the length of a, b, and c are given we can directly calculate the angle alpha with

Figure 2.3:
A = shoulder,
B = elbow,
C = hand target.



$alpha = acos((a * a - b * b - c * c)/(-2 * c * b)).$

Now the rotation from the shoulder towards the hand is calculated. This rotation is rotated away by the result of the above equation. The direction to bend the arm is chosen in a way that the upper arm seems natural and does not collide with the cockpit. The forearm's rotation is now trivial to calculate because the distance between the elbow and the hand target is exactly the length of the forearm. The mech arm now perfectly points its hand on the hand target position.

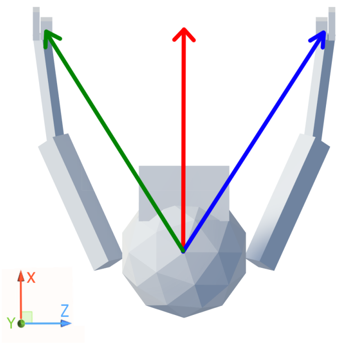
2.5 Mech orientation

During mech mode the user can control the direction the cockpit is facing. To determine the mech target rotation the following vectors are added up:

- Head forward,
- Head to right hand,
- Head to left hand,
- Current cockpit forward.

The y coordinate (the upwards axis) of all vectors is set to zero and then the vectors are normalized. This is done because the cockpit is supposed to only rotate around the y axis. The head forward vector is multiplied with 2.5, so it is weighted more than both hands together in order to keep the cockpit facing forward when both hands are behind the head. The vectors are then added up and are used as the new forward vector of the cockpit.

Figure 2.4:
Head forward = red,
Head to right hand = blue,
Head to left hand = green.



A problem that occurs when directly using the head forward vector like this is when the user looks more than 90° upwards or downwards. Then the cockpit suddenly turns in the wrong direction. To fix this the head forward vector is calculated as follows:

```
Vector3 headForward = Vector3.zero;
//When head.transform.forward.y is positive the head.transform.down is added to the forward
//When head.transform.forward.y is negative the head.transform.up is added to the forward
headForward = head.transform.forward − head.transform.up ∗ head.transform.forward.y;
headForward.y = 0;
headForward.Normalize();
```

This ensures that the cockpit points in the direction the head would point to when it wasn't rotated upwards or downwards.

Chapter 3

Physics based Control

This chapter will first explain the main benefits of the physics based approach to control and movement. Then the early version of controlling the Rigidbodies with Unity joints will be explained. Afterwards the PID controller [3] and how they are used in Unity will be explained in detail, followed by some methods that improve the stability and responsiveness of the simulation. Lastly the interaction with the environment will be shortly explained.

3.1 Motivation

The normal approach to control the parts of the mech would be to just set the position and rotation according to the user input and some Inverse Kinematics algorithm. By doing this one can exactly control what the mech itself should be doing, but the interaction with the environment can be problematic since a physics engine can not properly resolve the collisions of the teleporting colliders with no velocity.

The physics based approach that is chosen for this project is to give all mech parts Rigidbodies and colliders that are only controlled by adding forces to them. Unity can now accurately simulate all parts physics and also resolve their collision correctly. The main advantage we get from this is the realistic physical behavior that can easily be used to interact with the environment. It also solves the problem that often occurs in other VR fighting games that use the players hand position to directly control weapons. Then the weapon can be moved arbitrarily fast and goes through enemies

[3]PID controller: proportional-integral-derivative controller

or the environment. With the physics based approach the mech hands or weapons will collide with walls and also transfer the impact through the whole mech body.

The main challenge of this approach is to make the movement and controls to be responsive and fast.

3.2 Unity Joints

The first version to control the orientation of the limbs used Unity's Configurable Joints[4]. They were used to connect the different mech parts and fix their positions relatively to each other, and their Angular Drive was used to rotate the limbs to the result of our Inverse Kinematics algorithm. The three options to configure the strength of the drive is to set the "Position Spring", which determines the strength of the drive, the "Position Damper", which basically slows down everything, and the "Maximum Force".

The main problem of this is that the drive doesn't slow down enough before the objects rotation is close to the target, it just uses less strength in that direction, so it always overshoots. To improve the response time and reduce the overshoot of the Drive the Position Spring is made higher when the angle was higher, and the Damper is increased the closer it got to the target. All number tweaking didn't help much as the result was not stable and responsive enough and always included some overshoot.

[4]See [2] for the Unity manual of Configurable Joint.

3.3 PID controller

An alternative way of adding force is the PID controller[5]. It is a control loop feedback mechanism that is often used in industrial control systems. The PID controller receives the the error between the current state and the target state as input. This error is the proportional (P) part. Then the integral (I) and derivative (D) of the error are then calculated in the PID controller. These three values are now multiplied with different factors and the result is added to the object as force. This done every physics timestep.

- The proportional force grows linearly the further the object is away from its target.
- The integral part grows with time when the object is constantly off the target. This is used to counteract gravity and helps against steady-state error.
- The derivative part adds additional force depending on the change between the target and the object. When the target moves away from the object, then additional force towards the target is added, and when the object already moves towards the target, force away from the target is added. This is very important because it allows the PID to stop on the target without overshooting.

The PID controller are a very easy to use and overall a very valuable tool, but for them able to do what they are supposed to do the factors for the proportional, integral, and derivative parts have to be well tuned, what can be really hard. It should also be noted that every part of the mech needs his own tuning and it eventually has to be retuned when some properties in the simulation are changed.

[5]PID controller: proportional-integral-derivative controller. This two sites are helpful in understanding how PID controller work: [7] and [3].

3.4 PID controller in Unity

Normally the PID controller is only used to control one axis, since normal robot arms have one motor per control axis. Because we do not have this restriction in Unity this project always uses three PID controller combined to one vector PID controller. So to control the final mech with ten Rigidbodies there are in total 30 PIDs updating every physics timesteps. This vector PID controller takes the error as input, calculates the derivative and integral of it. It then multiplies these three values with the corresponding factors.

The PIDs for position and velocity are very simple to implement, the current values are read out of the Rigidbody and are used to calculate the error. This error is the input for the vector PID controller. The result of the PID controller is then added to the controlled object with Rigidbody.AddForce()[6] . It can be beneficial to clamp the result to keep the forces in a reasonable pace.

The rotational PID is a bit more tricky to implement in Unity, because rotations are saved as Quaternions. The PID controller requires the difference between the target and the current rotation. But since Quaternion.eulerAngels[7] gives $(90, 0, 0)$ as output for the rotation $(90, 0, 0)$ and $(89, 180, 180)$ for the rotation $(91, 0, 0)$, we can't use this to calculate the error.

Instead we do the following:

- Calculate the cross product of the current forward vector and the target forward vector.
- Do the same for the right and the up vector and add all the cross products up.
- Use this as input for the PID controller.
- The result of the PID controller is directly added to the object with Rigidbody.AddTorque()[8].

In addition to this the integral of the PID controller is multiplied with 0.98 every physics timesteps to limit the additional force that can come from the integral. This is necessary when the arm collide with a solid object. Otherwise the integral would rise every physics timesteps, and in the moment the solid object is gone the arm

[6]See [4] for scripting reference of Rigidbody.AddForce().
[7]See [5] for scripting reference of Quaternion.eulerAngels.
[8]See [6] for scripting reference of Rigidbody.AddTorque().

would overshoot heavily. This PID does exactly what we want from a rotational PID controller. It is used for all rotating parts, meaning the arm and leg parts, and the cockpit.

It is very hard to visualize how the rotational PID controller works internally. The following diagrams are all from the first scenario. In four seconds the z axis of the hand is moved by four meters to the right and and back to the origin. In order to better explain how responsive the hand control is there is a positional PID controller (see in Figure 3.1)added to read out the values. This PID does not apply any forces. On the following page the diagrams from the upper arm and the forearm rotational PID controller are shown. The values in these diagrams are so small and have no unit because they are the crossproduct of multiple unit vectors added together. That is also the reason why they are so unintuitive and it is not possible to show the target or current rotation in the diagrams. For all diagrams there are 50 physics timesteps per second. The recording is stopped once the system is stable.

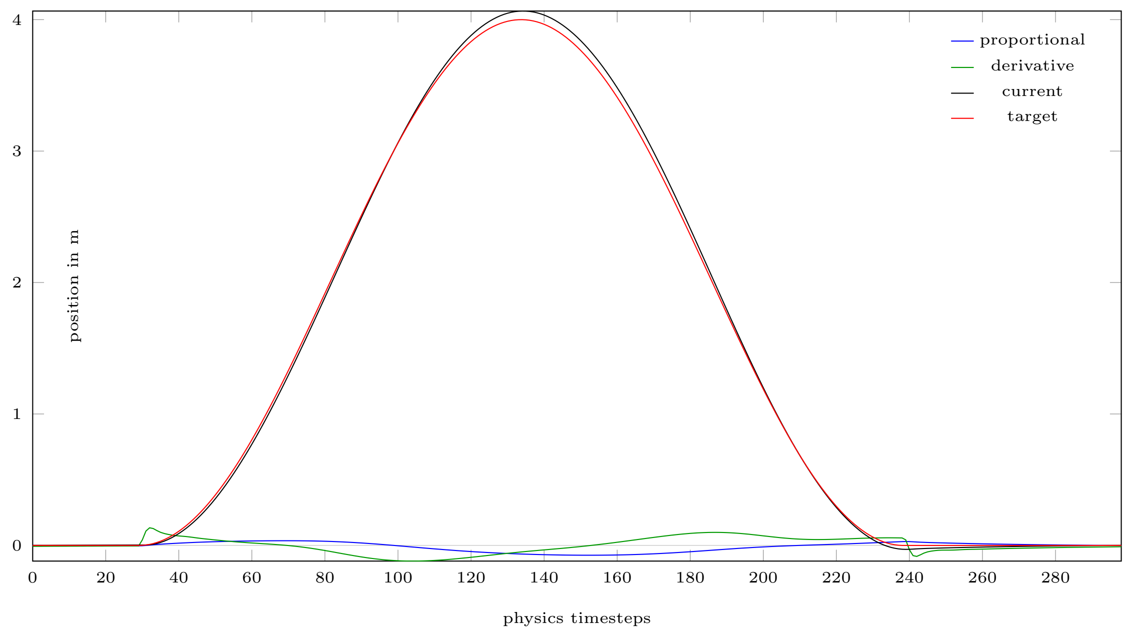


Figure 3.1: Z axis of a hand position PID. It was only used to read out the values, no forces were applied by this PID controller.

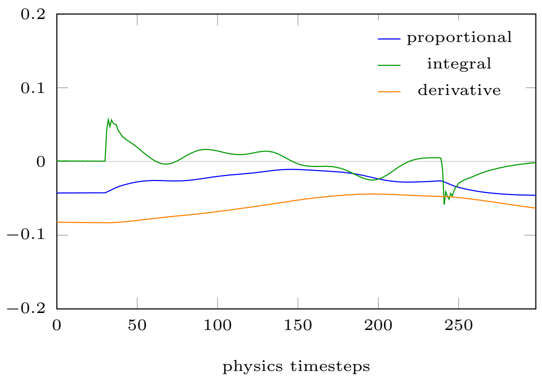


Figure 3.2: X axis of the upper arm rotational PID.

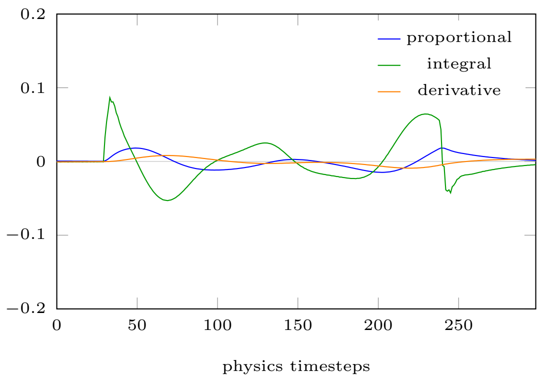


Figure 3.3: Y axis of the upper arm rotational PID.

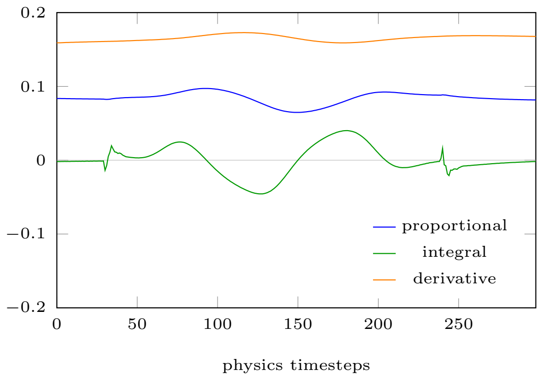


Figure 3.4: Z axis of the upper arm rotational PID.

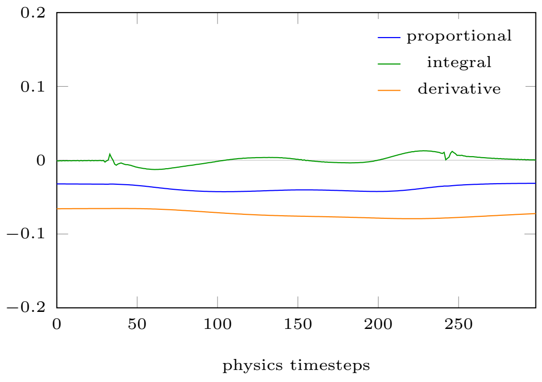


Figure 3.5: X axis of the forearm rotational PID.

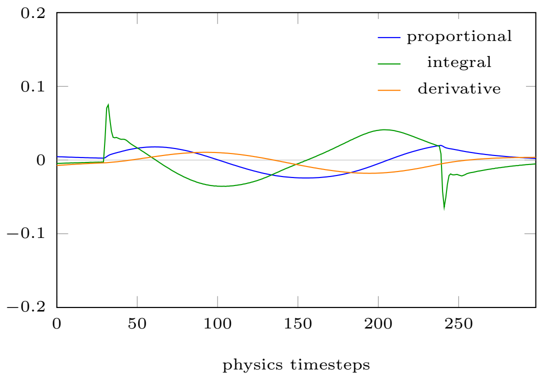


Figure 3.6: Y axis of the forearm rotational PID.

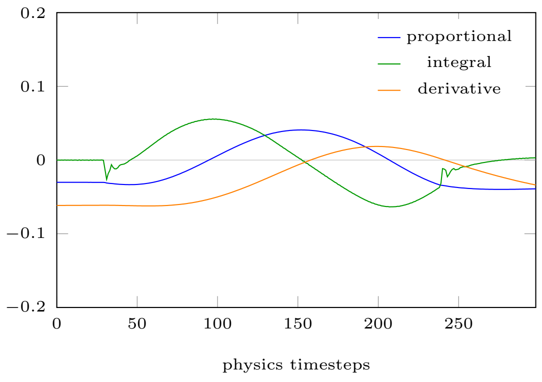


Figure 3.7: Z axis of the forearm rotational PID.

The same scenario is also tested on the last version with the Unity Joint drives. It is very obvious that they heavily overshoot, take longer to react, and need some time to stabilize.

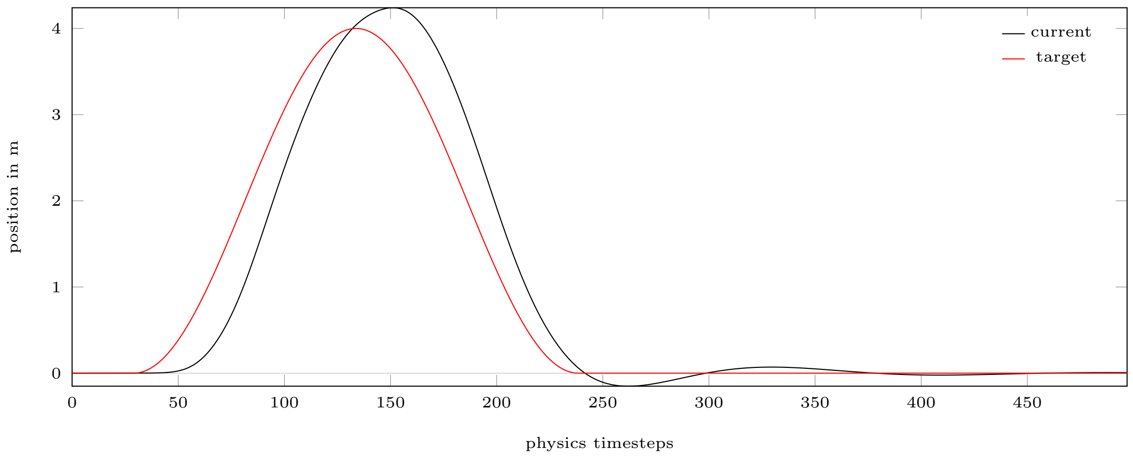


Figure 3.8: Same scenario as in Figure 3.1 but with the old joint motors.

The next two diagrams illustrate the second scenario. This time the target moves 2.5 m to the right, then five meter to the left, and after than back to the origin. Like the last scenario this is finished in four seconds. The main difference to the first scenario is that it starts and stops abruptly.

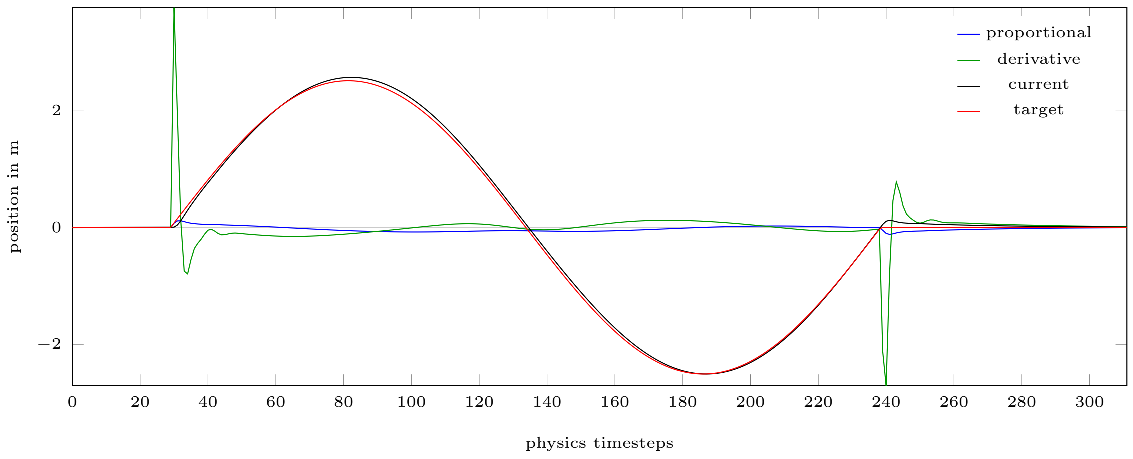


Figure 3.9: Scenario 2 with rotational PID controller. A position PID was used to track the effectiveness of the rotational PIDs. No forces where applied by position PID.

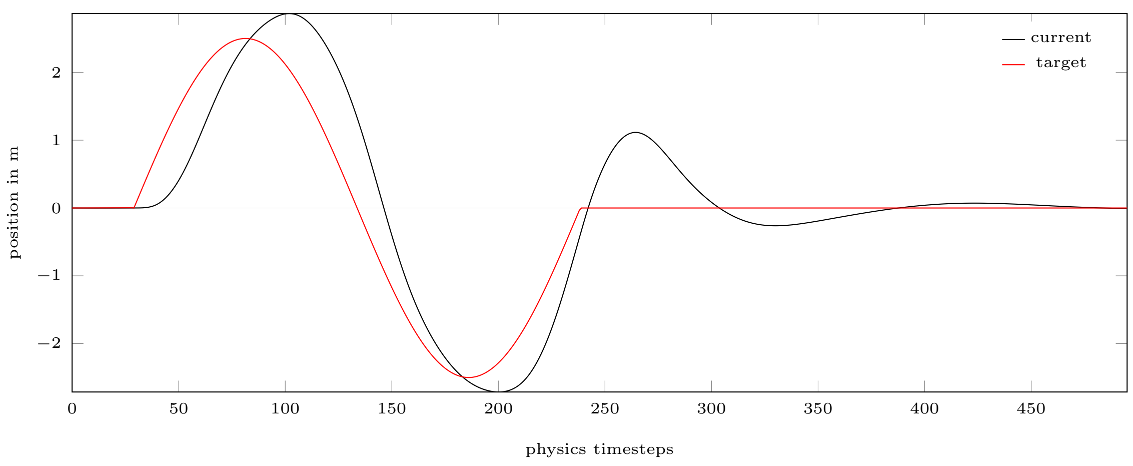


Figure 3.10: Scenario 2 with Joint motor.

3.5 Mech position control

There are several ways the mech force to move the mech can be calculated and applied. If the force were be applied to all mech parts, then it would act very weirdly when colliding with static objects, so all forces accelerating the mech are applied to the ball it stands on. This means the 185 kg heavy mech is moved by applying forces to a 15 kg heavy ball.

3.5.1 Position PID

Using a position PID we have to set a target position. We do not want to control the mech by pointing on the ground and letting it move there. Instead the target position was set in the direction the user pointed on the touchpad. The distance of the mech to the target position is controlled by the touchpad on the Vive controller. The further away the target is, the faster the mech moves. The main advantage of this technique is that the position of the user can be scaled and added to the position that comes from the touchpad control. This way the position of the user is directly translated to the mech so that the mech.

3.5.2 Velocity PID

With an velocity PID we can directly control the velocity the mech is supposed to have. The target velocity is set by the touchpad. This technique is more responsive and easier to control than the position PID, but we can not directly use the user's position to additionally change the mech position. This was solved by adding the velocity of the user to the target velocity. This does it's job but is not as precise as what is possible with the position PID. Overall the velocity PID with the touchpad is definitely more responsive than the position PID.

3.5.3 Angular Velocity PID

This is not really an angular velocity PID. It uses the same input as the velocity PID, but instead of applying the result as directional forces we now apply it as torque. The PID input is the velocity difference and the output is used to change the angular velocity. Even though the ball has very small touching ground for friction it accelerates well and is also able to stop on point, but it feels like the mech does not accelerate very smoothly. Overall it is the most physically realistic option, but it is harder to tune properly and seems slower to react.

3.5.4 Combined Velocity and Angular Velocity PID

In the final game the mech uses both a velocity PID and an angular velocity PID. This means that the ball is moved in the target direction, and additionally is rotated towards the target direction. This more stable and responsive than the other solutions.

Since the mech consists of a system of multiple Rigidbodies it takes some time to slow down once the target velocity is close to zero. Especially the cockpit lags a bit behind, and since the whole mech is tilted forward during the break the cockpit moves backwards once the mech feet stop moving.

Diagram 3.11 shows the mech accelerating and decelerating with the combined velocity and angular velocity PID to a target velocity of sine scaled by ten. The proportional value are multiplied with 35, and the derivative values with 0.6. The factor for the integral part is 0 so it is basically a PD controller.

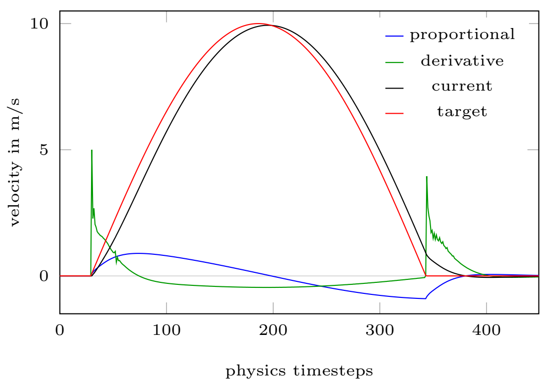


Figure 3.11: Accelerating the mech ball with the final combination of the velocity and angular velocity PID.

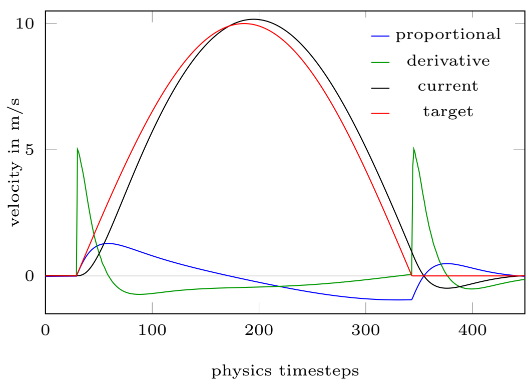


Figure 3.12: Same simulation as 3.11 but with tracked values from the cockpit. There are no direct acceleration forces to the cockpit, it receives all acceleration over the leg.

3.6 PID Tuning

In order for the the PID controllers to work, they must first be tuned. This means the three factors for the P, I, D terms have to be set to appropriate values.

PID tuning in Unity is a bit different than tuning a real PID controller, because Unity is a simplified physics environment without friction and other influences. It also directly applies the forces to the object. This results in the complication that we can not use the normal PID tuning concepts for the PID controller in Unity, but rather have to tune by estimating.

The general approach that was used when tuning the PIDs in this project was to first find a suitable ratio between the factor for P and D, such that it was not overshooting or oscillating. Then both values were increased as long as the system was still stable and not oscillating or overshooting. This can be done by eye but it is generally easier to plot the error, integral, and derivative in order to see how stable and fast the system is. The I factor can be set after the P and D parts are tuned, because it is often used to counteract the steady state errors and does not affect the performance of the controller too much if it is not set too high. When knowing the general dimensions of the factors, it is time to fine tune them. Table 3.1 can be used as a reference what to expect when increasing the factors for P, I, D separately, but due to the complexity of the system it is not always accurate.

Increase in	Rise time	Overshoot	Settling time	steady state error
P	Decrease	Increase	Small Change	Decrease
I	Decrease	Increase	Increase	Eliminate
D	Small Change	Decrease	Decrease	No Change

Table 3.1: This Table describes what one can expect when increasing the factors for the P , I, or D values [7].

To efficiently fine tune it is advised to change the values while the game is running and to experiment a bit. PID tuning is not very intuitive, trying out new values can sometimes help.

It is also important to test the PID tuning with a similar situation you would have in the game. If there is an instant position change, then it should be moved in big steps for the tuning. If there is continuous change like the Vive controller position, then the position should be modulated with sine or cosine.

3.7 Improving the physics simulation

This section will explain some methods that can improve the stability and overall reaction of the whole simulation. To make the simulation to behave like we want there is often a decision between realism and functionality. Since this project does not aim to create a fully realistic physics simulation there are some parts physically incorrect to improve the mech behavior.

- All the PID forces are applied with the force mode "Acceleration". This means that it is supposed to be applied every physics timestep and that the force is multiplied with the mass of the object. The main reason to use mass independent force is to be able to change the mass without retuning of the PID controller.
- One important aspect is to have adequate mass for all Rigidbodies. The mass of a single object does not really matter for our implementation of the PID controller since we use the mass independent force mode "Acceleration" to apply the forces. If there are multiple objects connected to each other, the relative mass of each object is very important, because if an object has ten times more mass it forces will also be ten times stronger, and it will be less affected by lighter objects. This is used for the cockpit, since it is reasonable to have more mass and to be less affected by it's arms, and it overall helps to achieve a stable cockpit orientation, what is very important to reduce motion sickness.
- It is also advised to keep the mass and size in a realistic scale, because objects with different sizes often behave differently in a physics engine. In this project the cockpit weights 100kg, each arm 25kg and the leg 35kg. Objects that weight about 1kg or less now behave strange and can be pushed into walls or the ground by the mech arms. Every object we want the mech to interact with should have realistic mass compared to its size, in order to have a realistic looking physics simulation.
- The gravity of every object can be changed. In this project the gravity is basically a force that the PID controllers have to fight against all the time. In a position PID the integral factor can counteract the gravity, but can cause questionable behavior with heavy overshooting when running against solid objects. In the earlier version of this project the gravity of the arms was deactivated because the joint motor was not able to counteract the force. This leads to

very floaty and slow falling, because only about half of the mech's mass is affected by gravity. In the final PID version every physics object is influenced by gravity, including the weapons the mech can pick up, without it being noticeable to the user. The gravity helps to achieve realistic behavior of the arm, so hitting an object from above adds extra impact due to gravity.

- The center of mass of Rigidbodies can be changed to custom values. Normally Unity calculates the center of mass and the inertia tensor at the start of the game depending on the colliders of the Rigidbody. This updates once the colliders change. When these values are changed with a script, then Unity does no longer update them, so we can set custom values at the start of the game to better control how the Rigidbodies behave. For the mech arm parts the center of mass is set to (0,0,0) because the way we add rotational force to them can not specify the position the rotation is supposed to be applied to. The new center is at their joints, where the force is supposed to be in order to maximize the effectiveness.
- The inertia tensor of the arm parts is set to (1,1,1) multiplied with their mass to have the same force on every axis. so it is not necessary to tune every axis independently. The inertia tensor of the cockpit was also set to custom values to reduce the tilting that the cockpit makes when accelerating forwards or sidewards, in order to improve the user experience and reduce motion sickness.
- When using a chain of multiple Rigidbodies, like the mech arms, it is beneficial to let every part try to reach it target rotation as if all other parts already have the right rotation. This means that the target rotation of the forearm is independent of the actual rotation of the upper arm. This is a very important step to eliminate vibration and oscillation between the parts.
- A precondition of fluent movement and animations is to also have fluent input. This can easily be achieved by applying Unity's Slerp[9] and Lerp[10] to the different inputs, in order to decrease or remove jitter induced by tracked hand or head position. This jitter does not necessarily come from inaccurate tracking, and mostly is only noticeable when the hand rotation is used to rotate a bigger object.
- With fast rotating Rigidbodies it is often necessary to increase the property

[9]Spherical interpolation between two vectors or quaternions [8].
[10]Linear interpolation between two vectors or quaternions [9].

Rigidbody.maxAngularVelocity[11]. This value limits the maximal angular velocity an object can have, and directly limits the turn rate by that. The default is relatively low, and even limits the rotational speed of the robot arms. After increasing it the mech was able to accelerate fluently and the arms responsiveness was improved. Diagram 3.13 and 3.14 show what effect this has for the ball the mech stands on. When increasing this value, it should not be too high because the physics engine becomes unstable when simulating objects that rotate to fast.

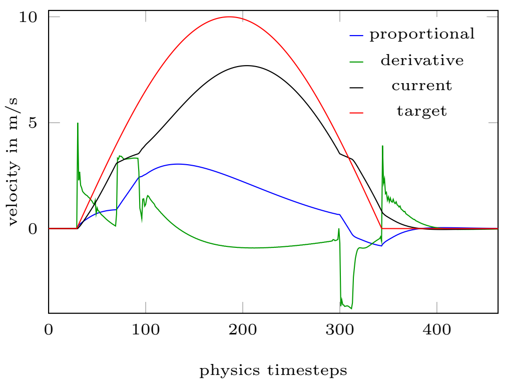


Figure 3.13: Same acceleration as 3.11 with the maxAngularVelocity default value of 7 instead of 25. At 3 m/s the rotation limit begins to slow the mech down.

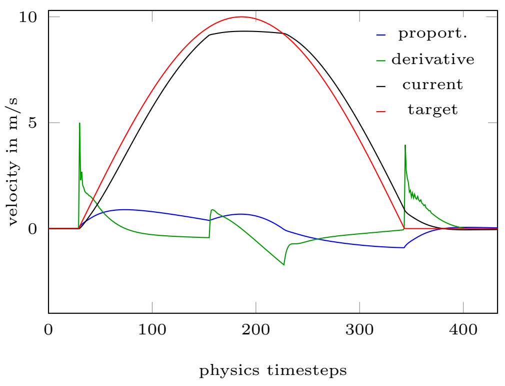


Figure 3.14: Also same acceleration as 3.11 but with maxAngularVelocity = 20 instead of 25. The ball can not turn faster so the mech can barley accelerate over 9.1 m/s.

- The PID controller gives the theoretical force necessary to achieve the target they are given, according to their tuning. This means they can give us very big values that the Unity physics engine could not reliable work with. One solution to this is to clamp the result values to a reasonable number, what will increase reaction time. It is advised to retune the PID controller after the clamp, and probably reduce the overall output when the results are nearly always the maximum value.
- If the simulation is still not fast or reactive enough, then it is possible to add forces that are extremely unrealistic but help to achieve the intended behavior. It would be possible to add an additional positional PID that directly moves the hand to the target position. This technique seems to be used by most physics based games that use physics controlled animations. Before using the

[11]See [10] for the scripting reference of Rigidbody.MaxAngularVelocity.

PID controller this idea was considered, but with the PID controller it is no longer necessary.

3.8 Interaction with the Environment

Thanks to the physics based approach on the mech control there is only little left to be done to achieve proper interaction with the environment. It is possible to pick up cubes with a mass up to 50 kg with the friction of the hands, and it is also possible to stack them accurately, or throw them around. The mech arms are also strong enough to climb ledges that are about twice as high as it's shoulders.

In addition to this natural way of manipulating the environment the user can also pick up objects with the trigger. If the object is no weapon, the hand is simply connected to the object by a fixed joint. If the object is a weapon, then the hand is moved to the position and orientation this weapon is supposed to be held before the fixed joint is created. Then the physics engine and PID controller repositions the hand back to where the user wishes it. This way the weapon is always held correctly, and as a user it felt better than teleporting the weapon to the hand or moving the weapon slowly in the hand. This can be improved by limiting the distance to the hilt where the weapon can be picked up, so it is not possible to pick a sword up at tip of the sword.

Chapter 4

Mech Design

Since the mech movement and control is physics based the design of the mech is not only about aesthetics, but rather about function. This chapter will explain the design process of the different mech versions. All models were created in Blender.

4.1 First Prototype

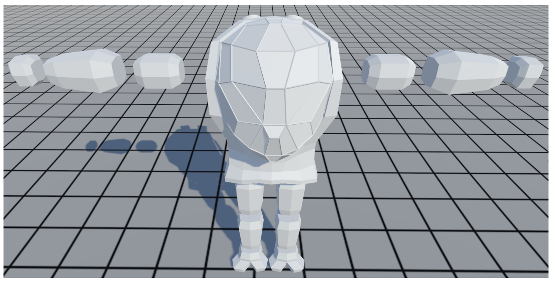


Figure 4.1: First prototype mech front view.

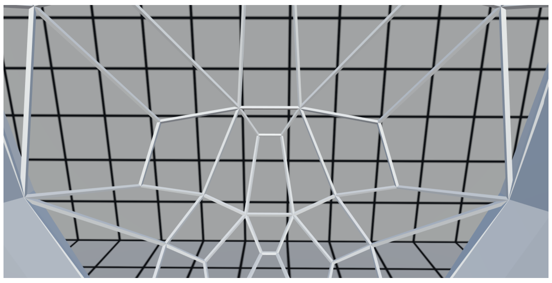


Figure 4.2: First prototype view from the inside.

The first model was a very early prototype where the main goal was to have a cockpit with enough space for the player to stand in and to reach out with his arms. It was also mainly used to help myself developing the concept how the mech arms move and can be controlled. The model was oriented on human proportions, but during tests it turned out that the arms were to short too feel useful. This is due to the cockpit being very long, so the area where you can interact with the hands is limited.

The main point to improve on this was to increase the cockpit field of view and to reduce the amount of bars in the glass front.

4.2 Second Prototype

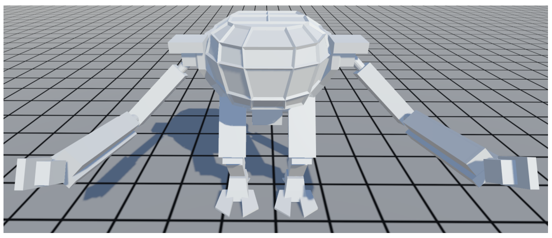


Figure 4.3: Second prototype from the front.

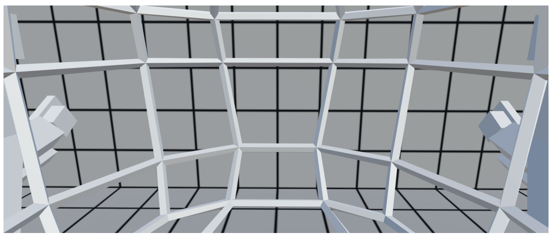


Figure 4.4: Inside view of the second prototype.

The whole design of this version was centered around the cockpit. The idea was to make it big enough such that the player can take one step away from the middle and reach out with his hand and still be inside the cockpit. This aspect worked very well, the player was able to stand on a circle with 1.25 m radius and had about 50 cm extra space for the hands to reach out. During testing it turned out that two meter from the ground to the ceiling was not enough, even though the test person was 1.8 m tall. The reason for this is probably that we are used to walk in rooms with a ceiling height of about 2.5 m, so it felt very limiting. The user could also easily reach over the ceiling.

The arm parts are now connected to each other, and the user can see the orientation of each part. The mech arms are longer than the arms of a human with the same height, because the front of the cockpit is about 1.5 m in front of the shoulders. This ensures that the user still has a big area in front of the mech where he can interact.

This model has legs, but since it is not feasible to use them with this physics based concept they had no physical interaction. To keep the cockpit hovering over the ground the ideal height had to be calculated with a script and then a position PID was used to keep the the cockpit at this height.

4.3 Final Mech

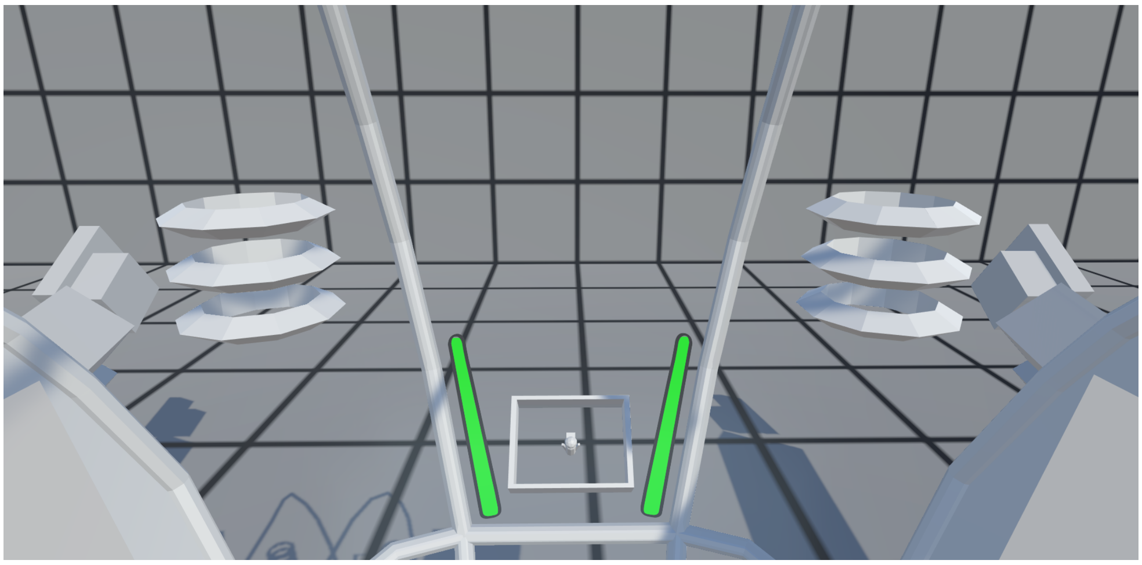


Figure 4.5: View from inside the cockpit of the final mech.

For the final design the goal was again to reduce the number of bars in the glass front of the cockpit. The glass is not like the previous prototypes out of many small glass parts with bars in the edges, but out of one big arched glass part with 3 bars in it. The bars are left in to give the user the feeling that he is inside a cockpit in order to reduce motion sickness.

With the decision to set the player position to the middle of the cockpit during the mech mode, the size of the cockpit is only dictated by the arm length of the user. This cockpit is like the first prototype which was designed to fit when the user looks in the same direction as the cockpit faces, and it is not necessary to give the user a good sight no matter where he stands.

Even though the cockpit is only 2.1 meter high it does not feel as restricting as the last prototype, probably because there are no horizontal bars in front of the users face.

Another addition to this mech is a realistic movement option. Now, since the mech has one middle leg with a free spinning ball as his feet, it allows us to use the Unity physics and gravity to have the mech on the ground and have realistic, suitable behavior. The mech is moved by applying forces to the ball and is overall easier and more realistic to control than the old versions.

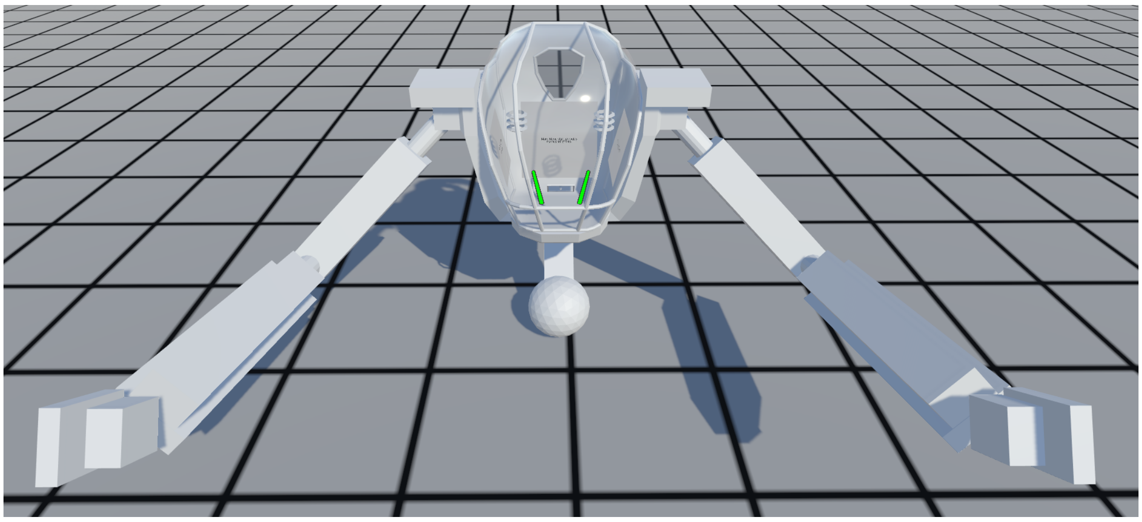


Figure 4.6: The final mech from the front.

In direct comparison to each other the mechs became smaller with every prototype, and the arms became longer. Especially the mech legs became smaller with every version since you do not see them in the cockpit, and reducing the distance to the ground creates more action and also feels more immersive. It also makes it easier to pick up objects from the ground.

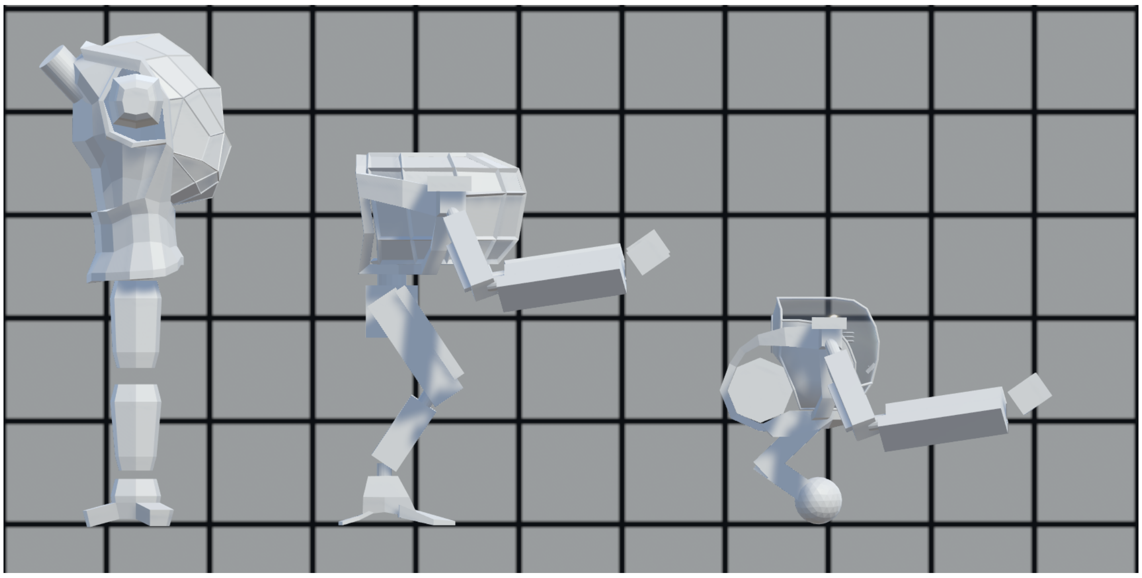


Figure 4.7: View of all three mechs from the side. The distance between the black lines is 2 m.

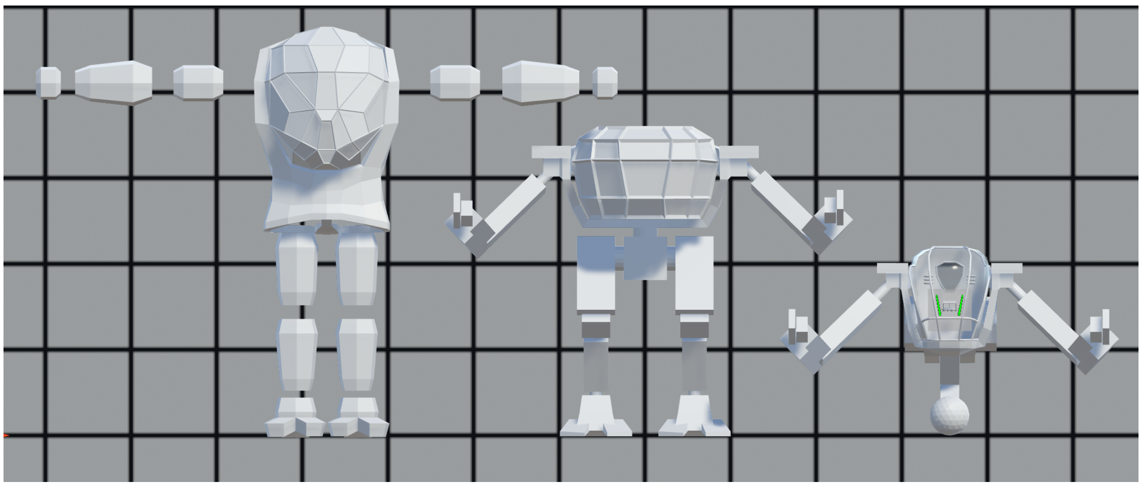


Figure 4.8: View of all 3three mechs from the front. The distance between the black lines is 2 m.

Chapter 5

Discussion

This chapter will discuss similar Virtual Reality projects, and then list some interesting ideas that could improve the current simulation.

5.1 Related Work

Overall there are very few virtual reality mech games or simulations on the market, even though parts of the Virtual Reality community really want them. During the development of this project the first high quality mech VR game "Archangel" was announced and later launched[12]. It has the same core idea of controlling the arms of a mech, but they took a different approach. The hands are used to fire an array of different weapons, but can only be controlled in a very limited space in front of the cockpit. It is intended to be a seated game, meaning that they do not utilize the position of the player in the real world, and the objective is to shoot enemies. The mech movement is not controlled by the player, but it rather follows a predetermined path. Archangel is an immersive story driven rail shooter.

There are also some other mech VR games in development by indie teams, but nothing I found gives the user control over the mech's hands.

The inverse kinematics that is used in this project is very simplified and only works for the specific scenario of arms and legs with two parts. Real inverse kinematics algorithms allow to simulate whole character models with many limbs that can consist of multiple joints. With this it would be possible to give the user a virtual

[12]Steam store page for Archangel: [11].

body. This is done in many multiplayer VR games but is always inaccurate because there are too many possible solutions with only tracking the head and hand positions.

5.2 Future Work

A feature I would like to add to the current miniature (Figure 2.2) is something that helps the user to untwist the Vive cable. What this means is that it would count how often the user turns around himself, and notify him how often in which direction he has to turn to receive the original rotation of the Vive cable.

The recently released Vive Tracker[13] would help to improve the simulation by giving us more information about the user. The back rotation could be used to set the cockpit rotation, and the forearms position and orientation would reduce our Inverse Kinematics problem to one solution.

An other very interesting addition would be the upcoming Knuckles controller[14]. This controller can measure how far the user closes each fingers individually. This could be used to accurately simulate the mech hand with fingers, and allow the user to realistically grab objects.

I tried to get automatic PID tuning to work, but I was not able to find any working algorithms that I could implement in reasonable time. The problem is again that the way I use the PID controller for the rotation is very unconventional. It would help immensely to have the PIDs tune themselves, especially when adding physics based enemies.

5.3 Conclusion

My goal was to create an immersive Virtual Reality mech simulation that uses realistic physics to control the mech parts. The biggest concern about this project was if the physics based arm control becomes responsive enough to be immersive and enjoyable. Now with the tuned PID controllers and continued improvements

[13]Vive Tracker: allows the user to track the position and orientation of any object. See [12].

[14]Knuckles Quick Start guide that describes the functions of the new controller. See [13] for a detailed description of the Knuckles Controller

over the course of the project the hands turned out extremely responsive. Even fast hand movements translate well to the mech.

The arms are even strong enough to climb ledges, and with the feature to connect the hand to fixed objects it is even possible to climb vertical walls and to hang under the roof.

Overall I am really happy with the current result, and will continue to add features to eventually make a real game out of the current physics sandbox.

If you would like to try this simulation yourself, you can send me an email at andi.leitner@tum.de

References

[1] Valve Corporation. "SteamVR Plugin". https://www.assetstore.unity3d.com/en/#!/content/32647

[2] Unity Technologies. "Configurable Joint". https://docs.unity3d.com/Manual/class-ConfigurableJoint.html

[3] Wikipedia. "PID controller — Wikipedia, The Free Encyclopedia". https://en.wikipedia.org/wiki/PID_controller

[4] Unity Technologies. "Rigidbody.AddForce". https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html

[5] Unity Technologies. "Quaternion.eulerAngel". https://docs.unity3d.com/ScriptReference/Quaternion-eulerAngles.html

[6] Unity Technologies. "Rigidbody.AddTorque". https://docs.unity3d.com/ScriptReference/Rigidbody.AddTorque.html

[7] Messner, B & Tilbury, D. "Introduction: PID Controller Design". http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID

[8] Unity Technologies. "Quaternion.Slerp". https://docs.unity3d.com/ScriptReference/Quaternion.Slerp.html

[9] Unity Technologies. "Vector3.Slerp". https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html

[10] Unity Technologies. "Rigidbody.maxAngularVelocity". https://docs.unity3d.com/ScriptReference/Rigidbody-maxAngularVelocity.html

[11] Skydancer Interactive. "Archangel". http://store.steampowered.com/app/553880/Archangel/

[12] HTC Corporation. "Vive Tracker". https://www.vive.com/eu/vive-tracker/

[13] Lawrence & JustJeff & Woodshop & jwmucha. "Kuckles Quick Start". https://steamcommunity.com/sharedfiles/filedetails/?id=943406651

All sites have been last accessed on September 10, 2017