



Linnæus University

School of Computer Science, Physics and Mathematics

Degree project

Production Cell Simulation Visualization in 3D



Author: Roger Valdeperas
Date: 2013-08-05
Subject: Computer Science
Level: Master
Course code: 5DV00E

Abstract

The thesis explains the development process of a production cell simulation in 3D implemented using Unity3D. The developed simulation communicates with existing control software and aims to test this control software in a 3D environment with physics simulation. The final result includes 3D models and also works as a visualization since it allows us to present the control system, and this visualization can be viewed using most web browsers. The thesis also includes a brief study and comparison between currently popular game engines to choose an appropriate option for this project.

This is a project in collaboration with a local company (ARiSA) and has a high practical relevance.

Key words: Unity, simulation, 3D, TCP communication, game engine

Table of content

1	Introduction	1
1.1	Background.....	1
1.2	Goal and goal criteria	1
1.3	Report structure	2
2	State of the art and basic technologies.....	3
2.1	Existing simulation.....	3
2.2	Alternative approaches	3
2.2.1	Unity3D	3
2.2.2	ShiVa 3D	4
2.2.3	Unreal Development Kit (UDK)	4
2.2.4	Minor candidates	5
2.3	Chosen engine	5
2.4	What does Unity provide.....	6
3	Architecture and design	8
3.1	Server-side vs. client-side control system	8
3.2	Communication protocol approaches	9
3.2.1	State pool on server	9
3.2.2	State polling.....	9
3.3	Chosen communication protocol.....	9
3.3.1	Receiving messages.....	10
3.3.2	Sending messages.....	11
3.4	Changing the control component.....	11
4	Implementation.....	13
4.1	Unity features	13
4.1.1	GameObject.....	13
4.1.2	Scripts	14
4.1.3	Editor extensions	15
4.1.4	Prefabs	16
4.1.5	Physics.....	16
4.1.6	Webplayer restrictions.....	17
4.2	Simulation devices.....	18
4.2.1	Scene setup	18
4.2.2	Belts	18
4.2.3	Cranes	20
4.2.4	Workspaces.....	23
4.3	TCP sockets in Unity3D	24
4.4	Cameras and GUI controls	25
4.5	Models and lighting	27
4.6	Installation	29
4.7	Error simulations	29
4.7.1	Physics errors.....	29
4.7.2	Connection errors	29
4.7.3	Control system related errors.....	30
5	Evaluation.....	31
5.1	Communication in real-time with control software.....	31
5.2	Online visualization in browser.....	31
5.3	Realistic	31
5.4	Interactive	31
6	Conclusion.....	33

6.1	Future Work.....	35
	References	36
	Appendix A Device references.....	38

1 Introduction

The project described in this report is about the development of a 3D online visualization of a production cell that communicates with existing control software. The visualization aims to test the control software in a 3D environment with physics simulation. The final project will also be able to present the control software since it will include 3D models and will have implemented an online visualization through web browser.

Some different approaches are discussed and the chosen one is described in more depth and detail.

1.1 Background

Automation (i.e. using machines commanded by control systems without human supervision) has been an important and fast-growing field during the last years, this is because it not only reduces product costs but also increases productivity and quality of the final product. This means the control systems need to be developed, but with them it is also needed to develop 3D simulations and visualizations to test and present such software products.

This project aims to simulate a production cell case study from Forschungszentrum Informatik (FZI) in Germany, this case study was previously described by Lewerentz and Lindner [20], [21]; and was also used by Lötzbeyer and Mühlfeld [22] to compare scheduling approaches. Later it was referenced by Zorzo et al. [18] to present a long term scheduling algorithm.

1.2 Goal and goal criteria

In this project the goal is to develop a 3D simulation for existing control software that will as well work as a 3D visualization.

The system we want to simulate represents a production cell that consists of four workspaces that are used to process metallic pieces; two belts, a feeding one where the pieces come from, and a deposit one where the pieces must end up after being processed by any of the workspaces; and two cranes, one of them must move the pieces from the feeding belt to the workspaces, and the other has to grab them from the workspaces and put them on the deposit belt.

The goal is to provide a simulation serving as testing environment and demonstrator of the control software.

Goal criteria include:

- Communication in real-time with control software: the main goal is to have the simulation connected in real time to the control software, which will issue orders to the devices in the simulation.
- Online visualization in browser: by demand this visualization must be viewed online through a web browser.
- Realistic: for it to work as a 3D simulation and visualization it should be as realistic as possible by including:
 - Physics simulation: it needs to be affected by real-time physics
 - 3D models and lighting: as a visualization it should include attractive 3D models that fit the appearance of the represented objects in real life
- Interactive: it must provide the user some controls for the simulation and it needs to react fast and properly.

1.3 Report structure

This report starts discussing the state of the art and possible approaches to the problem and explains in detail the one chosen to develop the visualization (chapter 2). It also discusses some design approaches to implement the communication between both systems and the system architecture (chapter 3) and highlights the most relevant parts of the implementation of the simulation and its devices (chapter 4). The end of the report evaluates the final result by using the goal criteria (chapter 5). The conclusion summarizes the report and talks about future work (chapter 6).

2 State of the art and basic technologies

In this chapter we discuss which engines or libraries could have been used to approach this problem to ensure that all goal criteria can be reached and then there are presented the reasons that made me use Unity3D.

2.1 Existing simulation

Before going further to the alternative technologies that we can use to develop the new simulation we should take a look at the previous simulation. There exists a 2D simulation implemented in Java that was being used to test the control system before the simulation visualization in 3D was finished.

The 2D simulation is shown in Figure 2.1, it has been reworked because it is not realistic enough, but as you will see the device layout has been borrowed from it and it is actually the same used in the 3D version.

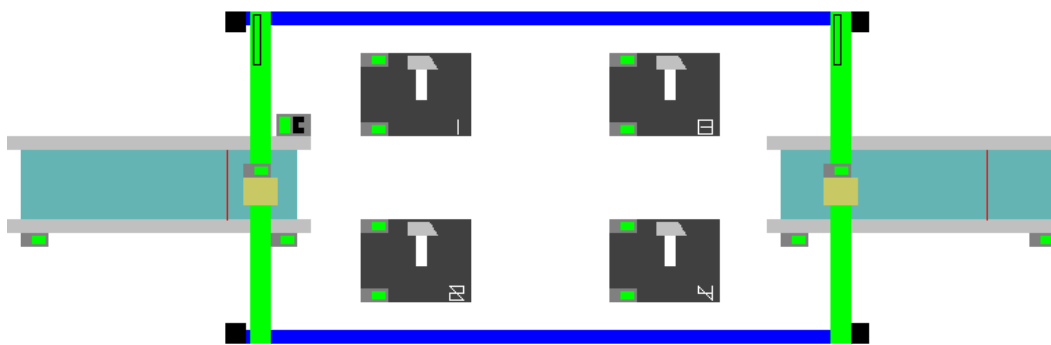


Figure 2.1 Java 2D simulation

2.2 Alternative approaches

Since there are no simulation-specific engines a good idea to develop a 3D simulation is probably to use a videogame engine since there are a lot of similarities between the visualization we aim to make and a videogame.

A game engine provides a framework that includes many features, usually including a rendering engine, a physics engine, input, sound, animation and networking (among others), and it usually includes an editor to make the scene creation easier.

Even though it would be possible to make my own engine, that would require a lot of time and developing it could be another whole thesis. Having this off the table, I decided to study the most widely used game engines nowadays and choose one that can fulfill all our goal criteria.

Here are the game engines that have been considered before starting this project.

2.2.1 Unity3D

Unity3D is one of the most widely used game engines right now as studied by Unity technologies [19], the main reasons are because it supports many platforms and its free version includes lots of useful features, but probably the most important is that thanks to the promotion that Unity technologies has made of it, it now has a really big community that supports it, it is really easy to find answers to anything and there is also a store with user-generated content that improves all the features it already has.

Considering the goal criteria:

- Communication in real-time with control software: Unity allows using TCP sockets and it is quite well documented. The Webplayer version has some restrictions but they are quite clear from their documentation page and they should not bring many problems.
- Online visualization in browser: Unity3D supports many platforms in its free version, Webplayer is one of them and it can run in most browsers, but the user needs to have the Unity plug-in installed to play.
- Realistic:
 - Physics simulation: Unity contains the NVIDIA® PhysX® Physics Engine which allows us to have pretty good physics simulation
 - 3D models and lighting: as a 3D game engine Unity allows us to import our own 3D models but, even though the free version includes lighting and many shading possibilities it does not include real-time shadows and some other advanced features.
- Interactive: as a game engine the interaction is one of the most consistent parts.

Unity3D would allow us to reach our goals and fulfill our goal criteria; the only drawback could be the lack of real-time shadows, which are not really necessary for the project, and we can still use baked shadows.

I have to point out too that I have used Unity3D on other projects before.

2.2.2 ShiVa 3D

ShiVa is really similar to Unity3D in many aspects; its free version can only build for web browsers, which is what we need. But, even though the documentation is clear enough, the community is really small and that makes it hard to solve the problems you may have while developing with it.

Related to the goal criteria:

- Communication in real-time with control software: as far as I know TCP sockets are not supported, ShiVa has some networking features, but this is not one of them, or if it is I could not find anyone explaining how to use it. Can be done with external plug-ins, but these are not easy to find either.
- Online visualization in browser: ShiVa can build for web browser with its free version (in fact it is the only platform you can build for with the free version), and, like Unity, requires its own plug-in installed on the user's computer.
- Realistic:
 - Physics simulation: ShiVa has physics simulation through ODE.
 - 3D models and lighting: ShiVa allows us to import our own models without many problems and, unlike Unity, has shadows in its free version.
- Interactive: as a game engine the interaction works pretty well too.

ShiVa seems to be a good option, but the TCP sockets would not be easy to implement, also the small community is an important point to have in mind.

2.2.3 Unreal Development Kit (UDK)

Unreal Development Kit (also known as UDK) is another candidate with free version and a good community. The main problem with UDK is that it is really focused on doing certain types of games and unless you have a really good understanding of how it

internally works it is quite hard to make anything that is not an FPS (First Person Shooter, specific type of game), and takes much more time than with the previous ones.

Regarding goal criteria:

- Communication in real-time with control software: it does not seem to have an easy way to do so using TCP sockets, they provide networking features but you cannot easily work with raw sockets.
- Online visualization in browser: UDK can build for flash, and it is free for non-commercial purposes.
- Realistic:
 - Physics simulation: like Unity, UDK includes NVIDIA® PhysX® Physics Engine
 - 3D models and lighting: UDK offers much better graphic results than Unity or ShiVa, with great effects, lighting and shadows. It also needs more requirements of course.
- Interactive: as a game engine the interaction works pretty well too.

UDK would give the visualization really good illumination and render effects, but again the TCP socket implementation seems to be the problem, it could take too much time to do it with UDK.

2.2.4 Minor candidates

These were the 3 major candidates considered, there are also some others which we did not really study in depth, but were also considered:

- CryEngine: Crytek's engine is well-known and has really interesting features, but it cannot build for web browsers.
- GameKit: this small engine does not build for browsers either, and its community seems to be quite small.
- Delta Engine: this one is an open source engine that could be promising too, but its current version cannot build for web browsers either (there is only a paid version that can build for Chrome), and its community seems to be really small too.
- Torque3D: another open source engine, it can build for web browser too, but the documentation seems to be incomplete and some features seem to be bugging a bit.
- Ogre3D: this is not really a game engine but a 3D library, it could be a solid candidate, but the fact that it is not a full engine makes everything harder and it takes much more time to create the scene (since there is no editor included) and you have to search the other libraries by your own.

2.3 Chosen engine

After studying some good candidates, the most used game engines nowadays, we have chosen Unity3D because among all engines is the one that makes fulfilling all the goal criteria easier and, furthermore, it is also more manageable for me to use since I have used it in previous projects.

Unity3D is a cross-platform game engine that can build applications for mobile devices (Android, and iOS), web browser, desktops (Windows, Mac, and Linux), Flash and consoles (Ps3, Xbox360, and Wii).

It has a scripting system built on Mono (open-source implementation of .Net) and the scripts can be written in C#, JavaScript (UnityScript) or Boo.

The graphics engine uses Direct3D, OpenGL or proprietary APIs depending on the targeted platform and it includes support for most shader techniques currently used, the Pro version also includes shadows and post-processing effects.

Its physics engine is NVIDIA® PhysX® Physics Engine.

Unity also has an Asset Store, where Unity technologies or any other developer can upload their own asset packages (models, textures, music, sound, effects, tutorials, projects, etc...) that can be downloaded for a low price, or sometimes even for free, decided by the developer.

Currently it counts with a pretty huge community that supports it, enhanced by the events created by Unity technologies around the world.

Even though Unity is not used by most of big game companies since big companies usually make their own engines, lots of Indie developers use it, especially to develop games for mobile devices.

2.4 What does Unity provide

Now that I have chosen an engine to work with it is time to discuss what does the framework provide and what do I need to develop.

This first list show which features I need for this project are already provided when using Unity:

- Physics simulation, triggers and collision detection: Unity can simulate physics and provides ways to detect collisions between physics objects. It also includes a special type of colliders that are triggers, which detect when would they collide with another body but do not actually collide with anything.
- Renderer and light simulation: of course, as a game engine it includes a graphic engine that can show 3D models and simulate lights with support for built-in or own-created shaders.
- Basic TCP interface (Mono): since the scripting language is based on Mono, which is an implementation of .Net, we can use the basic classes TcpClient and TcpListeners to implement the TCP connection.
- Webplayer build: as explained before building a Webplayer version, which can be executed from a browser, is possible in most game engines, including Unity. It does have some limitations, especially when using TCP sockets, but we will sort it out.
- GUI controls: Unity also provides ways to add GUI (Graphic User Interface) elements to your viewports, and it includes basic controls like buttons, sliders, labels, etc... and if it were necessary I could also implement my own controls.
- Animation support: this is barely used in this project, but Unity provides support for animation of 3D models, either imported from other software or made inside Unity itself.

This second list includes the parts not provided by Unity and that I will have to develop:

- Devices behavior: the main part that I have to implement is the behavior of each one of the devices in the simulation:
 - Crane: the crane is a device that needs to be able to move a magnet on the horizontal plane, and move it up and down to grab and drop the pieces.

- Electromagnet: the magnet needs to attract the pieces like a magnet would do it, and it should be possible to turn it on and off.
 - Belt: there must be belts capable of translating pieces over them.
 - Sensor: I will also need to implement a sensor to detect when pieces reach certain positions.
 - Workspaces: workspaces are devices that need to take a piece and spend some time processing it before returning the piece.
 - Piece feeding: I need to implement the generation of pieces during the simulation.
- 3D Models: Unity provides basic primitives but, of course, it does not provide the specific models we need for this project, the idea is to make some simple models to make it look as good as possible.

3 Architecture and design

In this section different system architectures and designs regarding the communication protocol are discussed, and at the end the one finally chosen and implemented is presented.

3.1 Server-side vs. client-side control system

Since the simulation must be viewed online, we had to discuss first which parts were server-side and which ones needed to be client-side.

The simulation has to be client-side, since is what the user's computer has to execute and show.

The control system, however, could be either server-side or client-side. Having it client-side seems to be the easiest one, since we just need to give both applications to the user and he just has to run them and watch the simulation but that would not reach our goal criteria of having real-time communication. Also, since the control system is a Java application and the simulation is on Unity Webplayer that would mean that we would need two independent applications on the user computer; actually, we would need three since we would need another application responding to the Unity Webplayer policy fetching (see point 4.1.6 Webplayer restrictions for more information).

Finally, for the reasons stated above, we decided to have the control system server-side, hiding it from the user, who will only see the simulation on his browser, as shown in Figure 3.1.

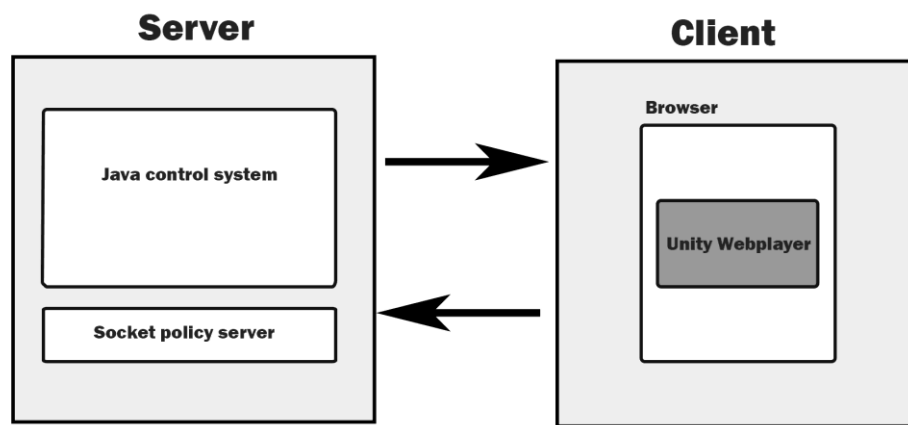


Figure 3.1 Chosen architecture, with the control system server-side and the simulation client-side

We also decided to build the control system as a multi-client server, capable of controlling many parallel simulation instances instead of having as many control system instances as simulations for efficiency reasons. So the following approaches aim to build this multi-instance control system.

Once this have been decided we need to discuss how are the TCP sockets going to be used, we decided to have 2 TCP socket connections, both ports hosted by the control system; one used to send messages from the control software to the simulations, and the other to send messages the other way. To keep it simple all clients will be connected to these 2 ports.

3.2 Communication protocol approaches

Choosing the communication protocol is really important in this project, especially because we need to communicate two independent applications (the control system and the simulation) made by different people using different technologies.

To make it work with only 2 ports for all clients we decided to use an id system: when a simulation starts it sends a message notifying the control system, which will answer with another message including an instance id, from then on every time the simulation wants to send a message to the control system it has to include its id into the message so the control system knows which simulation the messages comes from; and every time the control system wants to command an action to a device it will also include the id of the target instance so the others will ignore the message and only the instance with the corresponding id will forward the message to the target device.

3.2.1 State pool on server

The first alternative consists of a state pool on server, the idea here is that the states of all simulations are stored in the control system, and every time the state of a device in the simulation changes, the simulation only needs to send a message notifying the change to the control system, which will update the information it has about this simulation state. The control system can send messages to the simulation at any time to command the devices, which will react as soon as the message reaches the simulation.

This approach may consume some memory on the server since it has to store the state of all the running simulations, but it is a very good option to keep the amount of data sent relatively low.

3.2.2 State polling

An alternative approach is that instead of real-time signals we use a polling system, which means that the simulation never sends any message unless it has been asked to.

When the control system wants to attend an instance it first sends a message asking for the state of it, and when it receives the answer it checks the state and, if it is convenient, it can send an order to any devices of the simulation.

This approach saves memory on the server, but it implies sending a lot of data from client to server, and also makes the simulation less responsive since the changes are not notified instantly, they are not received until the server checks the corresponding instance. This means that this approach would not be able to communicate in real-time, and that is one of our goal criteria.

3.3 Chosen communication protocol

The implemented architecture for the final result is the state pool on server since it allows us to have multiple instances of the simulation and keeps the amount of data sent relatively low. This solution could bring problems if it had to support a large amount of simultaneous users, but since this is not the case and we do not expect more than 3-5 simultaneous users we will not dig deeper into that.

Now that we have chosen the architecture and decided how our communication protocol should be we will explain the details the communication flow. It has been decided bearing in mind the restrictions of the Webplayer, which are explained in section 4.1.6.

The process will work as follows:

1. The control software opens both ports and waits for incoming connections.

2. The simulation starts by fetching the socket security policy to the server.
3. The server answers the fetching with the policy, allowing the Webplayer to connect through sockets.
4. The simulation sends an initialization message to the control software.
5. The control software answers the initialization message accepting it and giving the instance a new ID.
6. When the simulation receives the message, it stores its ID. The ID will be sent along with every message to the control software so it will know which instance is sending the message.
7. The communication is now established and the simulation can be started.
8. From here the simulation will send signals for every change of state of any device and the control software will send signals to command the devices on the simulation.

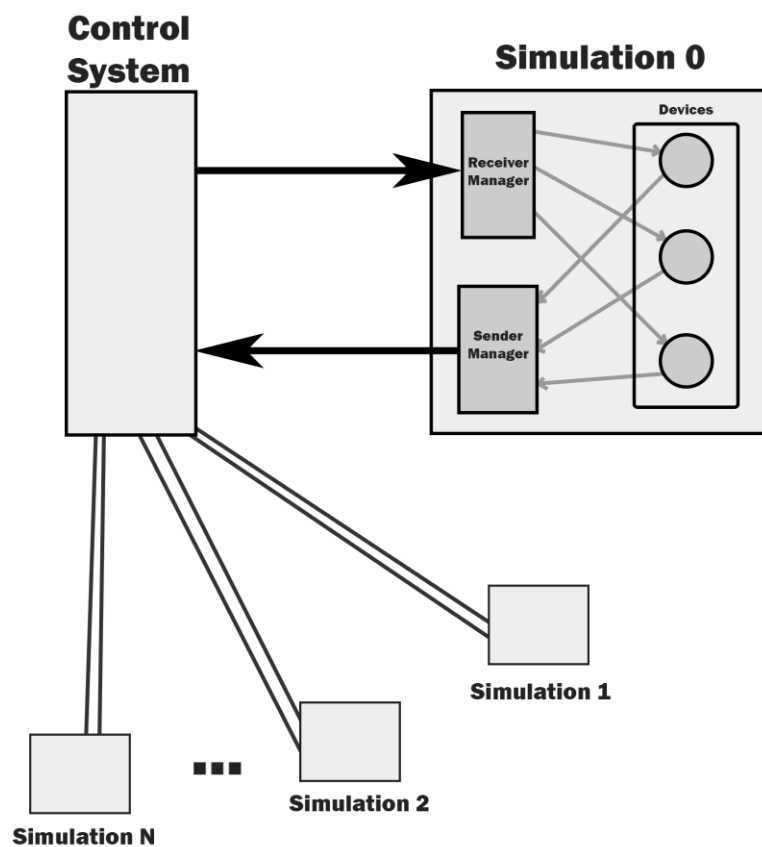


Figure 3.2 Communication sketch

3.3.1 Receiving messages

To receive messages first the simulation connects to the designated port and then it just processes the message every time it finds a new one, there will be a `ReceivingManager` that processes the messages. The `ReceivingManager` has to know all the devices so it can send every message to the proper device.

This is the list of messages accepted by the receiver:

- `MOVE_[Y]_PORTAL_TO_FEED_BELT`
- `MOVE_[Y]_PORTAL_TO_TRAY_BELT`

- [Y]_PORTAL_MOVE_TO_WORKSPACE_[X]
- [Y]_PORTAL_MOVE_TO_REGION_[Z]
- [Y]_PORTAL_TURN_OFF_MAGNET
- [Y]_PORTAL_TURN_ON_MAGNET
- [Y]_PORTAL_UP
- [Y]_PORTAL_DOWN
- WORKSPACE_[X]_START_PROCESSING

When a message is received the ReceivingManager will first check if the id attached to the message is its own, otherwise it will ignore the message.

3.3.2 Sending messages

To send messages the simulation first connects to the designated port.

I will have a SendingManager which works as a singleton with a static function that can be used to send messages from any other class, so when the simulation is running, each device can send messages by using this static function when their state change.

The SendingManager is in charge to add the ID to every message before sending it to the control software.

This is the list of messages that are going to be sent by the simulation:

- ITEM_IN_PICKING_POSITION
- ITEM_AT_TRAY_BELT
- [Y]_PORTAL_AT_FEED_BELT
- [Y] _PORTAL_AT_TRAY_BELT
- [Y] _PORTAL_ARRIVED_AT_FREE_WORKSPACE
- [Y] _PORTAL_IS_DOWN
- [Y]_PORTAL_IS_UP
- [Y]_PORTALT_AT_REGION_[Z]
- ITEM_PROCESSED_AT_WORKSPACE_[X]
- ITEM_AT_WORKSPACE_[X]

3.4 Changing the control component

This section explains how the simulation could be reused with another control system, since it works completely independent and could be used to test another control system with the same scenario.

To make another control server it would first need to host 2 TCP socket connections, one would be used to send messages from the simulation to the control system and the other would work in the other direction, from the control system to the simulation.

Then it will have to wait for incoming connections until a client connect to these 2 ports and sends a message *SIMULATION_INITIALIZED*, when the control systems receives this message it has to decide an id for this instance and answer with *id:SIMULATION_START*, at this moment the simulation will store this id and will start the simulation itself, from then on all messages targeting this specific simulation must include this id at the beginning (*id:MESSAGE*) so all other simulations will ignore it (since they are all receiving them, because we are using the same two ports for all simulations); and every time the simulation sends a message to the control system they will also include their id at the beginning (*id:MESSAGE*) to let the control system know where the message come from.

From this point the simulation will send to the control system the messages listed in section 3.3.2 when the state of the devices change and the control system can send the messages listed in 3.3.1 at any moment to interact with the simulation.

It is important to note that the state of each simulation instance must be stored by the control system.

When a simulation is closed it will send a message *SIMULATION_STOPPED*, so the control system knows that he can remove this client from the active list, this message, however, could not be sent in some occasions, if the simulation is not properly closed, so it might be important to add a timeout for inactive clients.

4 Implementation

In this chapter we explain the basics of the implementation of the simulation in Unity3D.

First there is a section explaining some basics of Unity; next I present how these features have been used to do the simulation: how the sockets are implemented, and how the main devices have been done.

The final section briefly explains the development of the 3D models and the shadow baking in Unity.

4.1 Unity features

This section explains the main features of Unity, but does not talk about the specific implementation of the project. In it there are explained the basics of the software.

Unity is a game engine that relies a lot on its user-friendly interface; most actions (but scripting) can be done with just the mouse and the most important region of the interface is the 3D view of the scene, as shown in Figure 4.1.

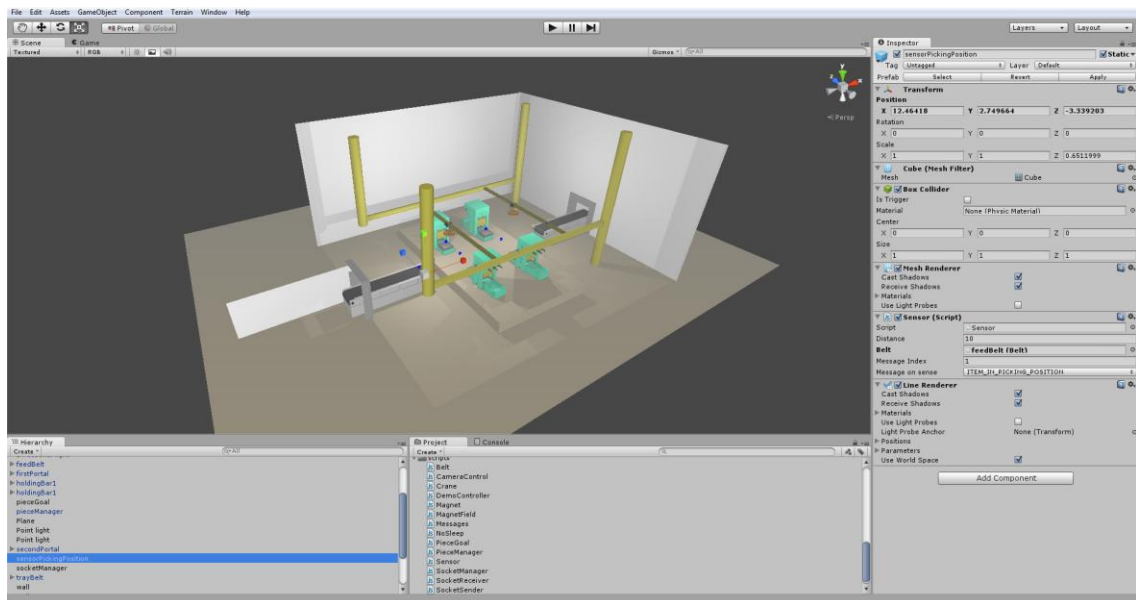


Figure 4.1 Unity interface

4.1.1 GameObject

The Unity basic class is called `GameObject` and it represents any entity in a scene. A `GameObject` can represent a camera, a character, a particle system, a light or anything else you can think of.

The idea behind the `GameObject` is that it is implemented as a container of Components; a Component is a feature of a `GameObject`, and it can be anything: a collider, a mesh, a physics body, an audio source, an effect, a script, etc...

So, by adding Components to a `GameObject` and setting their properties you define an entity for your project.

In Figure 4.2 you can see a `GameObject` with some Components in it, each one with its own properties. The specific `GameObject` shown has a Transform Component, a Mesh Filter Component, a Collider Component, a Mesh Renderer Component, a Rigidbody Component, and a Script Component, in this case using a script written in JavaScript called "Belt".

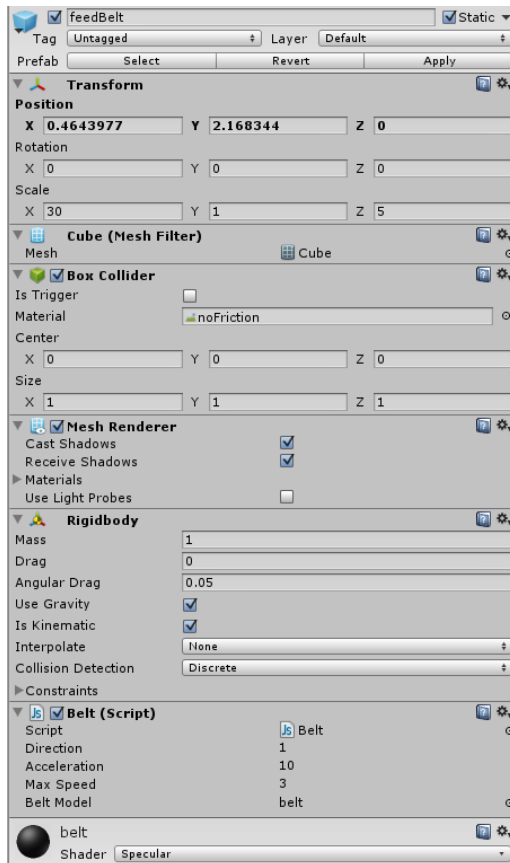


Figure 4.2 GameObject structure

And, of course, there are many more Components available in Unity.

4.1.2 Scripts

As stated before Unity3D has a scripting system built in Mono and can be written in many languages, in this project I have used JavaScript, sometimes also called UnityScript since it has some differences from the original JavaScript.

In Unity, as stated in the previous section, scripts are Components that can be added to a GameObject to make it perform any action; one GameObject can have as many scripts as you want.

A script in Unity actually represents a JavaScript class, with its own variables and functions. Public variables will appear as editable properties of the Component when added to a GameObject, so even though you can put an initial value, it can be changed on each GameObject. This allows us to reuse scripts by just setting these properties for each one.

When programming in Unity is quite usual to put many public variables, this is really useful to test your code, since these public variables not only allow us to set an initial value, they can even be changed during runtime, so it can help you to find the proper value of a variable while executing the project.

For example, in this Figure 4.3 you can see a script and in Figure 4.4 you can see how it is going to be shown when added as a Component in a GameObject.

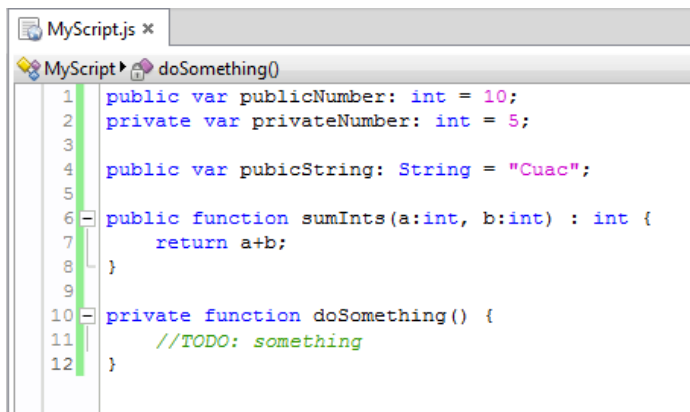


Figure 4.3 Example script, in JavaScript

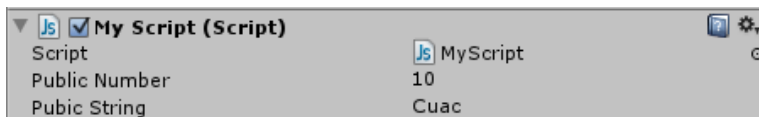


Figure 4.4 Result of adding the script from Figure 4.3 to a GameObject, showing the public variables as editable fields

From any script you can access to the other Components of the GameObject containing it, for example you can get or set the object position by using *transform.position*; or you could also set or get the velocity of the physics body with *rigidbody.velocity*; you could also call a public function of another script inside the same GameObject by doing *GetComponent(MyScript).sumInts(10, 5);*.

Components from other GameObjects can be accessed as well, for example, if we have a reference of a GameObject stored in a variable called *otherObject*, we could get or set its position by doing *otherObject.transform.position*; and all the other examples given above work for other objects too following the same pattern.

While programming in Unity it is also important to note that, by default, every class derives from a generic *MonoBehaviour* class, and it has many functions that are going to be called by Unity automatically that can be overridden; for example the function *Start()* is going to be called when the GameObject is created; the function *Update()* is going to be called once every frame; *OnCollisionEnter()* is going to be called when the GameObject collides with another; and a large etcetera.

Last thing that I think is important to note about Unity scripting is the delta time, which is actually used in practically all game engines.

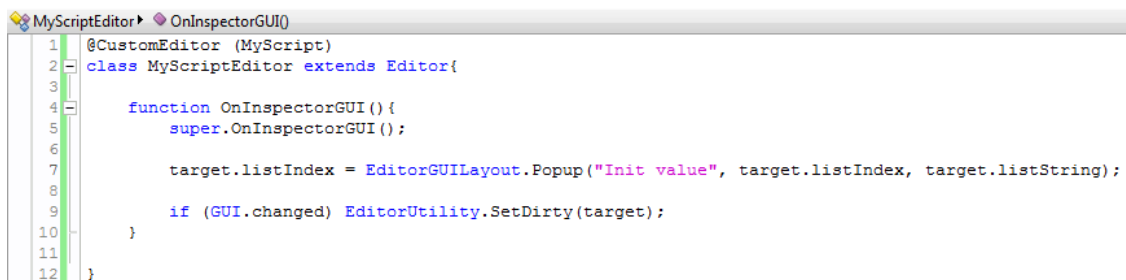
When programming with a game engine or any program that updates the state in real time it is important to know that you cannot be sure of how many times will the state be updated every second, in faster computers it will be updated more frequently than in slower ones, and depending on the other programs running on it that may vary during the execution. That is why if you want to move an object or update a time counter you cannot update a fixed amount every time you perform an update, you need to update it a different amount depending on how fast the state is being updated, and that is exactly what delta time is used for. The variable delta time stores the time in seconds that it took to complete the last update. For example, if we want to move an object 5 meters every second, we could do that on every update: *position += 5*deltaTime*.

4.1.3 Editor extensions

Unity also allows the programmers to extend the Editor, by creating their own Editor Windows or by changing how a public variable of a script is shown when added as a Component.

For this project I did not need any Editor Window, so I will not explain this feature in detail.

Customizing the properties setting is a useful tool in many occasions, for example, in case you need to set a string from a list, you can code an extension for the editor to make it look like a popup menu, this is going to be used in the project to select which messages are going to be sent for each device.



```
1 @CustomEditor (MyScript)
2 class MyScriptEditor extends Editor{
3
4     function OnInspectorGUI() {
5         super.OnInspectorGUI();
6
7         target.listIndex = EditorGUILayout.Popup("Init value", target.listIndex, target.listString);
8
9         if (GUI.changed) EditorUtility.SetDirty(target);
10    }
11
12 }
```

Figure 4.5 Script extending the Editor of the variables of another Script

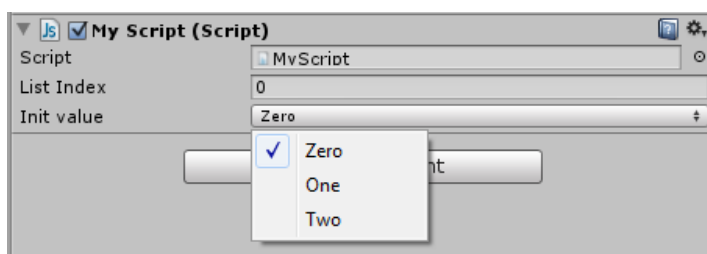


Figure 4.6 Result of the Editor extension shown in the Figure 4.5

4.1.4 Prefabs

Another important thing to have in mind when working with Unity is the concept of Prefab.

A Prefab in Unity is a stored GameObject; it is used when you want multiple instances of the same GameObject. When you make an instance of a Prefab, Unity keeps it linked to the Prefab, so if you change a Component or a property on the Prefab all its instances are going to be updated to include the same changes.

4.1.5 Physics

Since this is a simulation the physics engine is quite important for this project as well as understanding how Unity handles it.

In Unity physics bodies need to have a Component called Rigidbody to properly interact with the others. Rigidbodies have a linear velocity, angular velocity and other properties that can be read or written by code.

It is also important to note that the physics are updated much more often than the graphics, by default Unity updates the physics approximately 3 times for every graphics update, this means that when setting critical Rigidbody properties it needs to be done on the physics update, since doing it in the graphics update might end up causing strange physics behaviors.

When you add a Rigidbody Component to a GameObject you probably want to have a collider Component inside as well, this determines the shape that the object will use to collide (sphere, cube, capsule, etc...), which kind of physics material it is made of (with coefficients of friction and bounciness) and the option to make it a trigger. A trigger is a

special type of collider that does not actually collide with anything, but it detects when it would collide with another body.

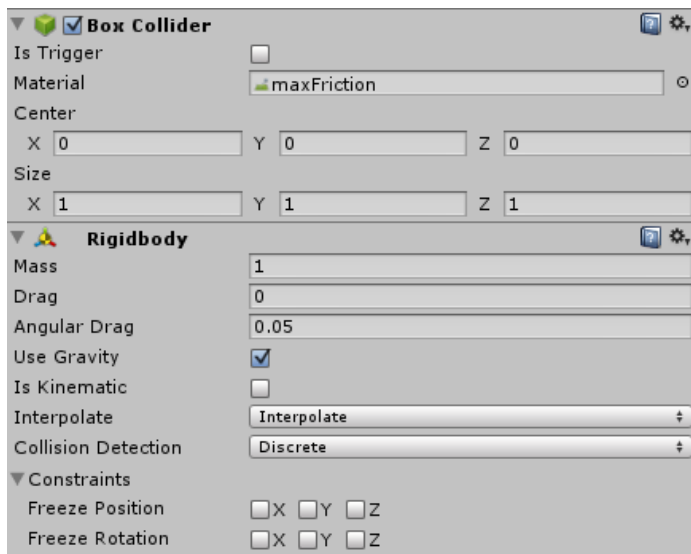


Figure 4.7 Collider and rigidbody Components in a GameObject, with their respective properties

Along with this, Unity also provides methods to detect collisions; in any script Component that extends from `MonoBehaviour` (all scripts do it by default) added to a GameObject with a collider Component you can use the provided functions to detect collisions or trigger-collisions:

- *OnCollisionEnter (col: Collision)*: is called for every other collider Component that began to contact with the target collider in the current frame.
- *OnCollisionStay (col: Collision)*: is called for every other collider Component that is colliding with the target collider in the current frame.
- *OnCollisionExit (col: Collision)*: is called for every other collider Component that was colliding with the target collider the previous frame, but is no longer colliding with it.
- *OnTriggerEnter (other: Collider)*: same as *OnCollisionEnter* but for triggers.
- *OnTriggerStay (other: Collider)*: same as *OnCollisionStay* but for triggers.
- *OnTriggerExit (other: Collider)*: same as *OnCollisionExit* but for triggers.

All these function have a parameter that contains information about the other collider.

The non-trigger functions also provide information about the collision, like the relative velocity or the contact points between both colliders and their collision normals. The trigger functions cannot provide this information since both bodies never collide, so both colliders usually overlap.

4.1.6 Webplayer restrictions

When building for Webplayer Unity adds some restrictions to keep a security sandbox, and among these restrictions there are some regarding the use of sockets.

The first important restriction is that a Webplayer cannot be a TCP Server, so the server needs to be on the control software, and the simulation will only act as TCP client.

The second restriction is that before connecting to the TCP server, the Webplayer needs to receive a security policy that has to be fetched manually. This means that the

server executing the control system needs to be ready to answer the fetching and the simulation needs to fetch the policy using the *Security.PrefetchSocketPolicy()* call that Unity provides.

For the server Unity also provides a little Mono executable that runs in background answering the fetches allowing the client to connect to certain ports.

4.2 Simulation devices

This section explains the implementation of the devices of the simulation.

4.2.1 Scene setup

The scene consists of: two belts, one that will be feeding the unprocessed pieces and another that will take the processed pieces; four workspaces, that will process the pieces; and two cranes, that will translate the pieces, each crane has access to one of the belt and both of them can access to all workspaces, so one crane will move the pieces from the feeding belt to the workspaces and the other crane will move the pieces from the workspaces to the deposit belt. There will also be one sensor (even though there are many more in the sketch shown in Figure 4.8) on the feeding belt to sense when a piece reaches the end of it, in this case the belt will stop and the control system will be notified that there is a piece ready.

The other sensors might provide more robustness to the system in case any error happen, but to keep it simple and do not give the control system the extra work of handling device errors we decided to remove them for this project.

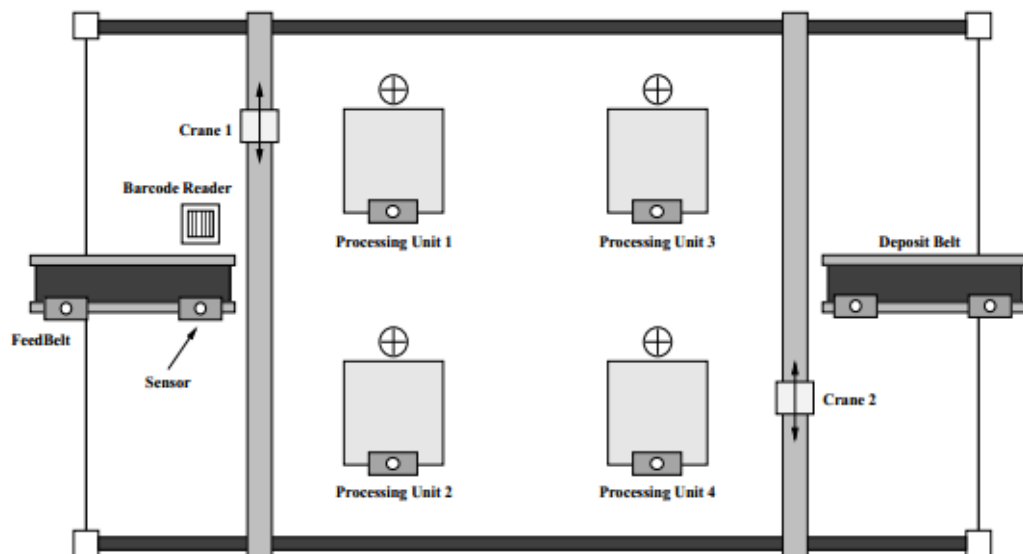


Figure 4.8 Sketch of the scenario setup – (A.F.Zorzo, L.A.Cassol, A.L. Nodari, L.A.Oliveira, L.R.Morais; *Long Term Scheduler for Real Time Industrial Installations*)

4.2.2 Belts

The belts are translating the pieces on them when they are switched on.

The feed belt is controlled by a sensor, which automatically stops it when a piece arrives to the end of it and starts the belt again when a crane takes that piece.

The deposit belt is always moving.

In Unity I implemented this behavior by setting a certain speed in a determined direction to all GameObjects colliding with the belt, to make it work the belts need to be completely frictionless so the objects can be freely moved over them, done inside the

OnCollisionStay() function, which is called every frame for every object currently colliding with the GameObject containing the script Component.

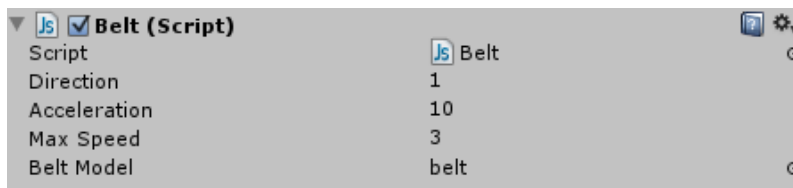


Figure 4.9 Belt script Component, with belt properties

The texture of the belt can be animated by code by changing the texture offset of the UV coordinates mapping to make it look like it is moving at the same speed that the pieces above it, can be done in the *Update()* function which is called once per frame.

This works because when you make a 3D model you also have to define how the 2D texture will be mapped when applied to the model; and Unity allows us to add some offset to this mapping. It also needs to have the repeat wrapping mode so the texture will be tiled and repeated infinitely.

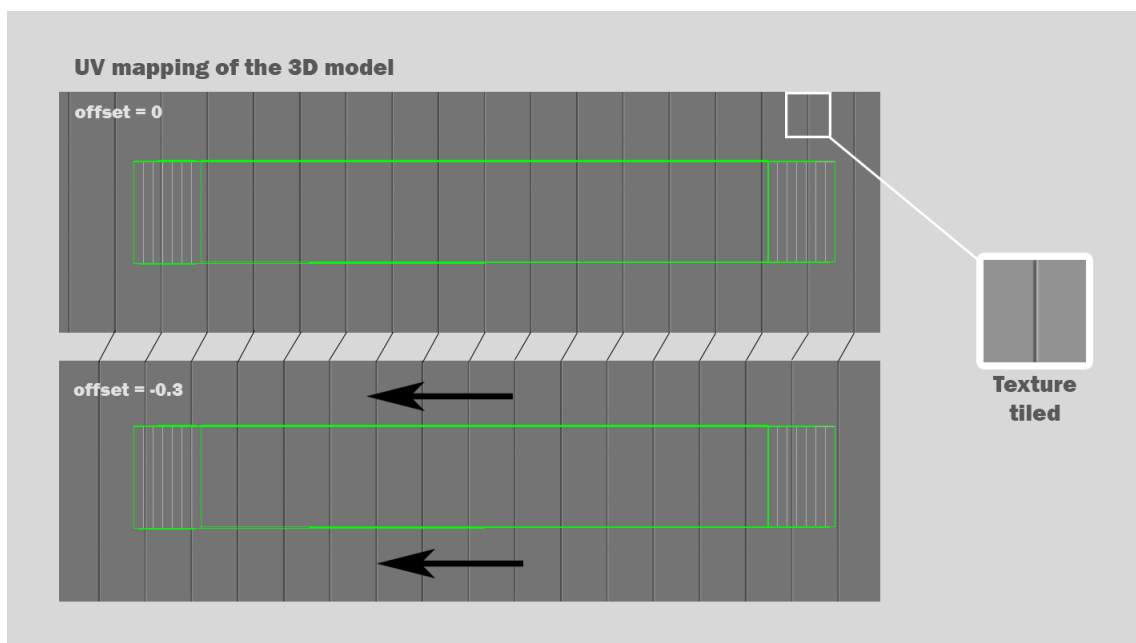


Figure 4.10 Belt model texture UV mapping, showing how the texture is tiled and repeated and how we can move the texture by setting the offset of the mapping.

For the sensor of the feed belt I used another function provided by Unity:

Physics.Raycast (origin: Vector3, direction: Vector3, out hitInfo: RaycastHit, distance:float): boolean;

This casts a ray from a given position to a certain direction and returns true if it hits any collider in the scene that is closer than a determined distance, in such case it also stores in the *hitInfo* variable information about the hit (like hit point or object hit, among other information). I used it with the following parameters:

Physics.Raycast(transform.position, transform.forward, hit, distance);

Where *transform.position* refers to the position of the GameObject containing the script Component and *transform.forward* refers to the local +Z axis of the same GameObject. The distance is a public variable to make the script more reusable and easier to test.

The Sensor script also contains two more public variables: the first is the belt that it must control; and the second is the message that should be sent to the control system when it senses anything. The selection of the message is done by using an extension of the Editor to make it easier to choose the message, since the messages are identified by integers is much better to make it a popup menu with the message themselves.

To make the sensor visible I used a Component called LineRenderer that allows you to draw a line from one point to another, simulating a laser.

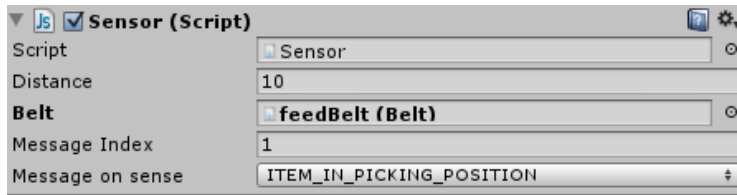


Figure 4.11 Sensor script Component

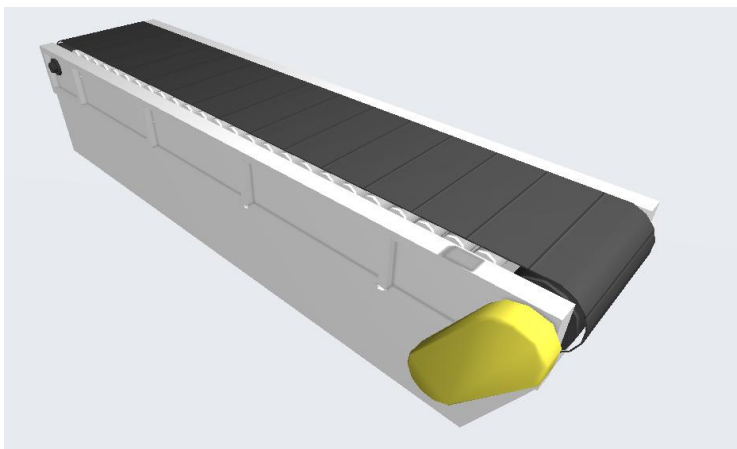


Figure 4.12 Belt model

4.2.3 Cranes

The cranes can grab pieces to translate them to another position of the scene; they consist of an electromagnet that can be switched on to grab a piece and switched off to drop it.

For this project the cranes need to be able to move to 8 determined positions: the two belts, the four workspaces, and 2 extra positions to wait when the other crane is working. The crane can go up and down as well, which should be done always before picking or dropping a piece.

The control system can command the crane to move to one of the 8 positions; to move up or down; or turn the electromagnet on or off.

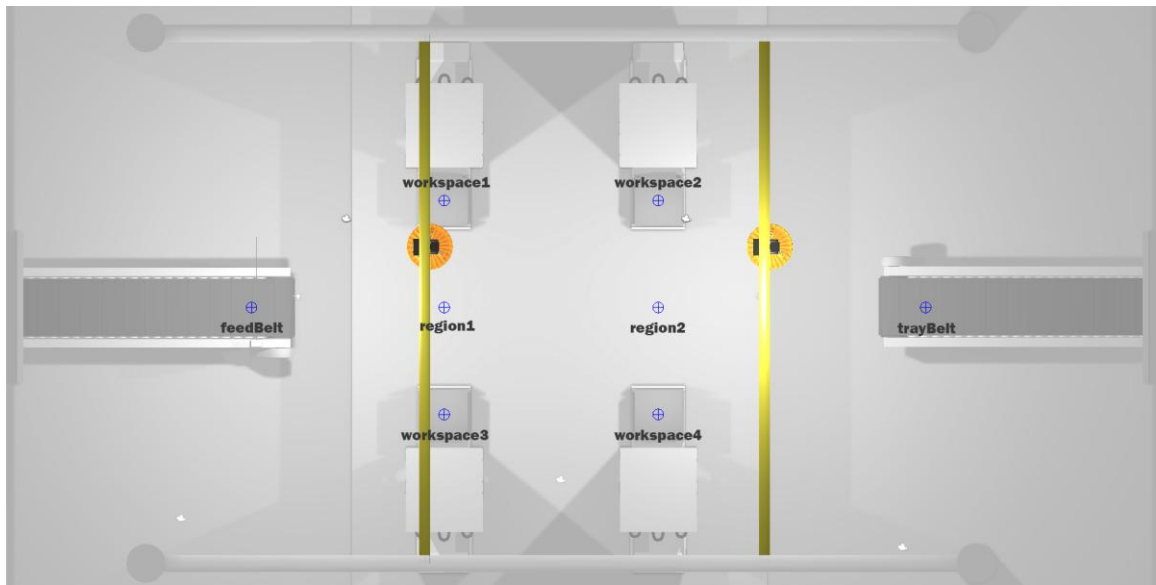


Figure 4.13 Crane positions

In Unity the crane object consists of 4 physics objects:

- Bar: the metallic bar can move horizontally through the X axis.
- MovingPiece: contains the piece that can move horizontally perpendicular to the bar movement (Z axis), we can think of it as if it was attached to the bar described above, so moving both objects the movingPiece can go to any position of the XZ plane.
- Cable: is attached to the movingPiece and can move up and down
- Magnet: is the last physics body, the one we are actually aiming to move, the magnet is attached to the cable to make it wobble a bit and make the movement more realistic.

In Unity the way to attach physics bodies is by using joints, joints allow you to connect 2 objects and let the physics engine move them as if they were physically connected, the most important joints supported in Unity are hinge joints, meaning that the two bodies connected have their relative movement constrained, allowing them only to rotate in one axis, like a hinge would do; fixed joints, meaning that they cannot move nor rotate relatively to the other body; and configurable joints, this is a completely configurable joint, allowing you to set which axis must be constrained in movement and which axis must be constrained in rotation.

To make the connections of the crane I used only configurable joints, since the basic types did not allow me to do what I was looking for. So, the magnet has a configurable joint allowing it only to rotate relatively to the cable; the cable has another configurable joint with only the movement in the horizontal plane restricted, so it can move up and down and rotate in all axis; the moving piece, on the other hand, is not using any joint since I decided it would be easier to just keep it attached to the bar by setting the same speed as the bar on the corresponding axis, this way I can ensure it does not make any weird movement due to the physics engine.

The script moves the crane with the acceleration and max speed determined by the public variables. I put all these variables public to make it easier to find the most appropriate values, since that would allow me to change them without modifying the code.

When the messageReceiver receives an order to move to a determined position it tells the crane to move there and it also tells it what message has to send when it reaches

the commanded position. For example, if the messageReceiver receives an order for the first crane to go to the workspace number 4, it will call the next function of the first crane:

```
goToPosition(pos: int, msgOnReach:String);
```

With the following parameters:

```
goToPosition(Crane.CRANEPOS_WORKSPACE_4,  
Messages.FIRST_PORTAL_ARRIVED_AT_FREE_WORKSPACE);
```

This will tell the crane to go to the forth workspace (according to its position list, as described in figure 4.8) and to send to the messageSender a message to tell the control system that the first portal arrived to a free workspace, the message will be sent when the crane reaches the indicated position, in this case the 4th workspace.

The cranes can also move up and down, but each crane has to send a different message to the control system when doing it, to do that I decided to let them know which message they have to send, since it will be always the same. I have the messages coded with integers, so just putting a number to refer to a message could be quite confusing, in occasions like this is when it is useful to make a little extension to the Editor so the messages can be chosen from a popup list.

When moving the magnet up and down I also scale the cables joining the magnet with the moving piece to make it look more realistic, otherwise the magnet would be floating.

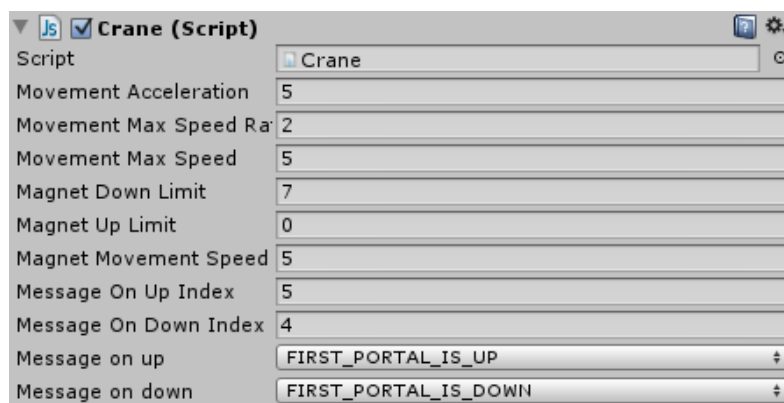


Figure 4.14 Crane script Component with the properties of the crane, using an extended Editor for the messages

To make the magnet work as a magnet, this means attracting pieces and holding those while the magnet is on, I did another object attached to the magnet that contains a trigger that acts as a magnetic field, positioned as shown in Figure 4.15. The idea is that while a piece is inside that trigger, I use the function *OnTriggerStay()* provided by Unity, which is called every frame for every GameObject staying inside the trigger, to add forces to the piece's physics body directing it to the center of the field depending on the distance (the closer the piece is to the center the stronger are the added forces). It is not fully realistic since the piece would not necessarily go to the center of the magnet in real life, but it makes everything go smoothly in the simulation since I can assure the piece will be dropped from the center of the magnet, the pieces do not go through the magnet because it contains a collider.

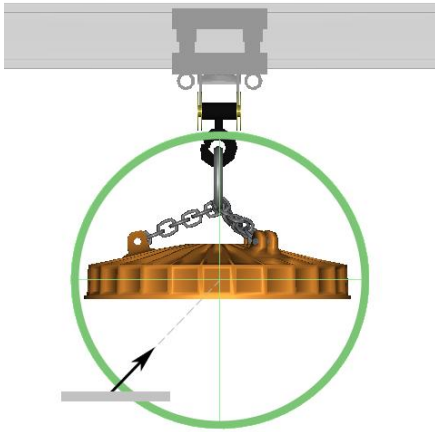


Figure 4.15 Magnet with the magnetic field trigger shown in green and the forces added to a piece that is partially inside the trigger



Figure 4.16 Crane model

4.2.4 Workspaces

The workspaces are the devices that have to process the pieces; they are supposed to be hydraulic presses for metal and they receive a piece on their platform, process it and put the piece back in the same platform so the crane can pick it up from the same position it was dropped on.

Workspaces are slightly different from other devices since they are animated models. This usually would not be a problem but in this case when the animation moves the platform it will also need to move the pieces over it, this means it has to interact with the physics bodies.

To do that I have created a collider object that will follow one of the bones of the animation, but attaching this collider to the bone is not a valid solution since it will move the object through its transform, which means it will not follow the physics simulation and the interaction between the collider and the pieces would not work properly; instead, I need to move the collider to the position of the bone every physics update, luckily Unity provides methods to do so. First there is a function that is called every physics update (which is performed many more times than the graphics update) called *FixedUpdate()*, inside I can put the code that I want execute every physics

update; second Unity provides a *rigidbody.MovePosition(position: Vector3)* that allows me to move a physics body to a certain position keeping a proper interaction with other physics bodies, so I can use this function to move the collider with the bone and have the collider following the animation. It is also important to let Unity know that you want this object animated on physics updates when importing it, instead of updating the animation along with the graphics as it is by default.

To animate in Unity we only need to have an Animation variable (set as a public variable in this case) and use *animation.Play(animation_name)*, where *animation_name* is a string that I chose when importing the model to Unity.

Workspaces also have a processing time that can be set as a property of the script Component, and a message on process property, that indicates which message will be sent to the control system once the press finishes processing a piece; this message has to be different for each one of the four workspaces to let the control system know which one is the workspace that has finished.

For the message sent I did another extension to make it easier to choose between the messages indexed by integers.

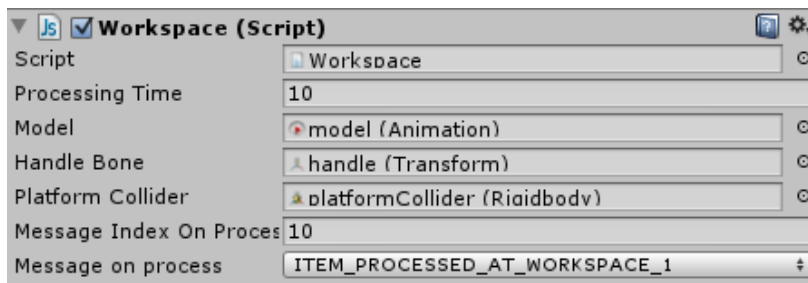


Figure 4.17 Workspace script Component, with extended editor view

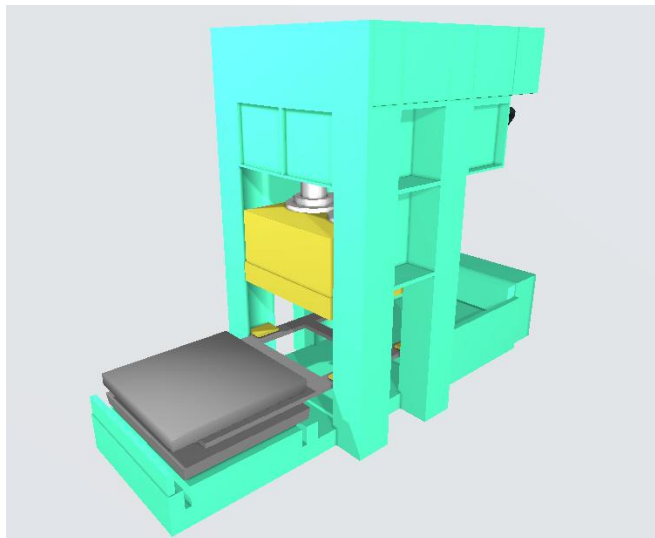


Figure 4.18 Workspace model

4.3 TCP sockets in Unity3D

Since Unity scripting system is built in Mono (implementation based on .Net) we can use the TCP network services provided by .Net, we could use the Socket class to manage the sockets, but .Net also provides two much more user-friendly interface socket communication classes, the TcpClient and TcpListener.

The communication works as explained in section 3.3, here I will just explain how it is implemented in the simulation.

The implementation then has been done using the `TcpClient` class for both receiving and sending messages, since both sockets are going to be hosted on the server.

To receive messages I have a `ReceivingManager`, it knows all the devices and forwards the messages it receives to the target device. The `ReceivingManager` uses the function `Start()` provided by Unity, which is called just once when the `GameObject` is created, to search all the devices in the simulation and store them to be able to send them messages when it is needed.

To send messages there is also a `SendingManager`, but it works completely different, what the sender does is just provide a static function that can be called from everywhere to allow any device to send a message whenever they need to, `SendingManager` works as a singleton so when a device wants to send a message notifying a state change it only needs to call `SendingManager.sendMessage(messageId: int)`.

All messages are stored in a class that contains all messages associated with their own ids, this way I could change the string sent without modifying any other file.

4.4 Cameras and GUI controls

To make the simulation more interactive I added support for different cameras and some GUI controls to let the user choose the Camera he wants to use; there are also GUI controls that allow the user to set the simulation speed and the piece feeding frequency.

I also added another group of GUI controls to set the properties of the devices.

Unity provides support for GUI with the basic controls you could find in any other GUI control set: buttons, sliders, text fields, etc...

I am using Unity's *SelectionGrids* for some controls, which allow me to show a grid of buttons to let the user choose between many alternatives. For other controls I am using *HorizontalSliders*, which allow the user to choose a value inside a determined range.

Unity also includes a way to customize the GUI styles and provides some example styles; in this project I am using a slightly modified version of one of the example styles (called "Amiga500").

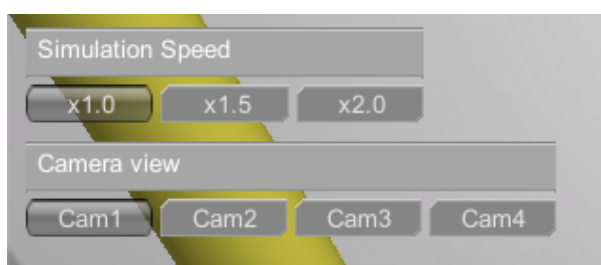


Figure 4.19 Graphic User Interface, simulation speed and camera view controls, done with `SelectionGrids`

To control the cameras I have all of them positioned in the scene and I just enable/disable them to render just from one of them. I attached one of them to one of the magnets so it follows it and gives a different point of view.

The simulation speed can be controlled through a variable from Unity's `Time` class called `timeScale`; for example, setting `timeScale=2` will execute everything to double speed and setting it to `timeScale=0.5` will make the simulation run to half speed. However, scaling the time to high values may cause problems with the physics simulation since the computer might not be able to perform physics updates so fast. For when this happens Unity allows us to decide if physics frames should be skipped or if

we want the simulation to slow down to take its time to compute everything. For this project I have decided to slow down the whole simulation when this happens, since when I tried skipping frames there were lots of bugs with the physics, pieces were dropped sometimes or cranes went crazy; this, however, prevents me of giving the user the alternative to put the simulation at more than x2.0 speed since most computers will not be able to compute all the physics updates and it will end up capping it at x2.0 anyway. In slow computers, x2.0 speed might not even be real x2.0 speed; it will be the fastest the computer can compute all physics.

The user can also set some properties of the simulation devices, like cranes speed, workspaces processing time and piece feeding frequency.

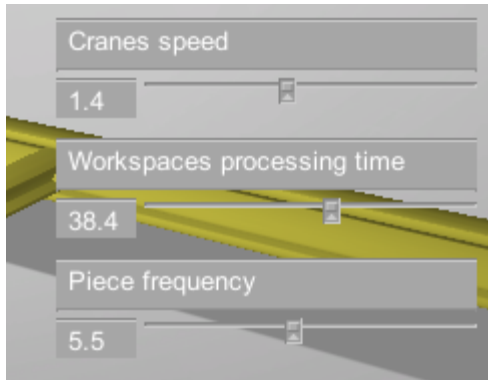


Figure 4.20 Graphic User Interface, cranes speed, workspaces processing time and piece frequency can be changed with sliders

The idea of letting the user change the cranes speed, the workspaces processing time and the piece feeding frequency aims to allow him to test a determined control system in different scenarios.

I set the processing time and movement speed of the cranes by just changing the corresponding variables of the devices.

To test a determined control system I also added some statistics that compute the efficiency and charge of each workspace.

In the statistics I compute the percentage of total pieces that have been processed in a certain workspace, to see if the system correctly splits the work charge between the workspaces; and I also compute the average time each workspace have been inactive between two processed pieces, this means the time that it was not processing anything and it was waiting for a new piece to arrive.



Figure 4.21 Statistics window, with percentage of use and average inactivity time between pieces for each workspace

Since the statistics window is quite big and it covers an important part of the visualization I have decided to let the user decide when it is shown or hidden by pressing a button.



Figure 4.22 Button to show the statistics window when it is hidden

I also considered another way to implement the GUIs which is also possible in Unity; it consists on having the buttons or other controls in HTML and call functions inside Unity using JavaScript, this way I would have in the browser a viewport showing the Unity project and some HTML controls around. I tried it but I finally decided to have them inside Unity because this way they look more integrated with the simulation.

4.5 Models and lighting

During the development process all the tests had been made on a scene with really simple objects, 3D primitives, and since one of the goal criteria is making it look good and realistic to be presented I decided to spend some time on modeling and lighting the objects of the scene.

In Figure 4.23 you can see the first version, fully functional but not really good looking. For this version I was using 3D primitives provided by Unity (cubes and cylinders) and I only had the necessary objects to code, as you can see all objects are floating, and the magnets are not properly connected to the cranes.

Also, I was using colors to know the state of each device.

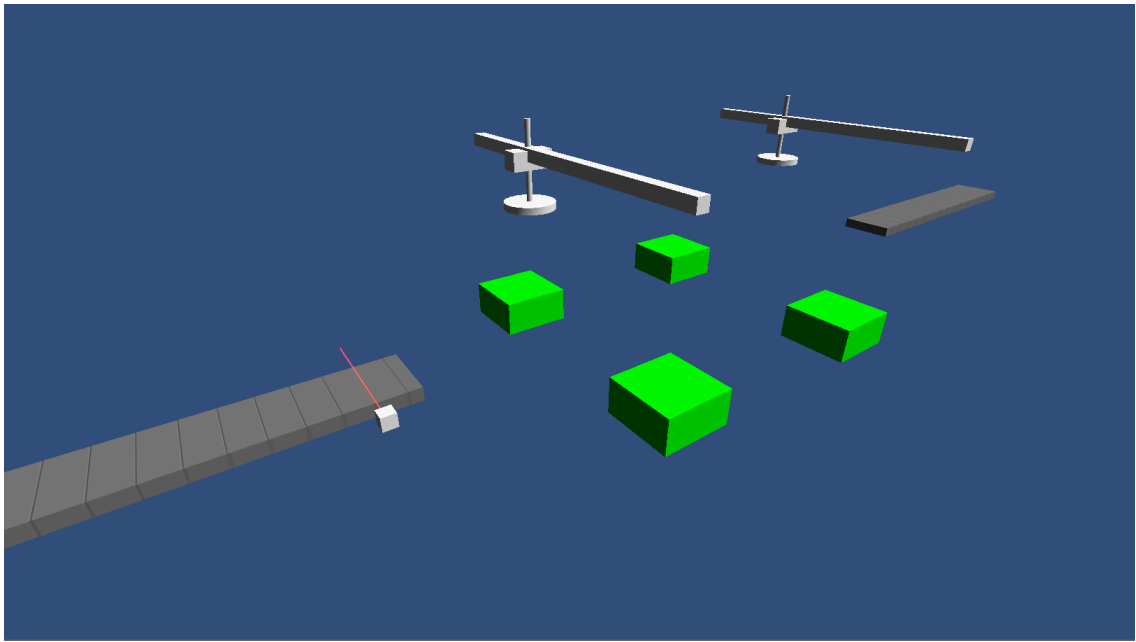


Figure 4.23 First version of the scene, with 3D primitives to represent the devices

I looked for some references of actual cranes, belts and presses (see appendix A for more information) and I made some models with 3ds Max 2013, which I had some experience with, and added them into the scene to make it look a bit better, as seen in Figure 4.24. In this version I worked on some materials for the objects, but I did not want to spend too much time making the textures since I thought it was looking good enough with simple materials with plain colors.

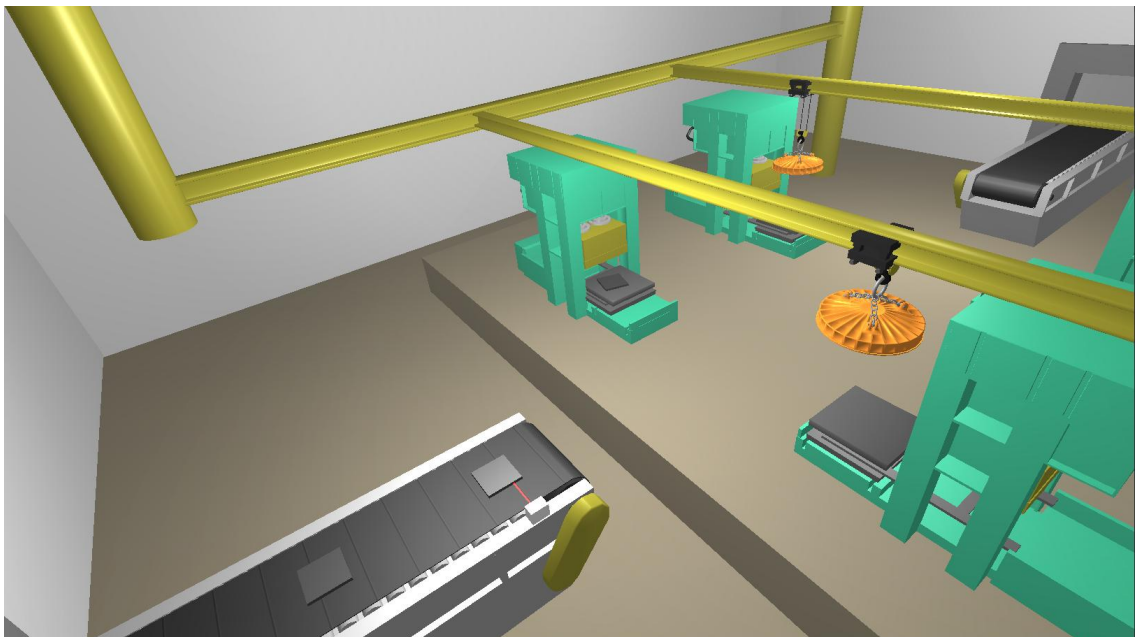


Figure 4.24 Version of the scene with 3D models

And even though the Unity free version does not support real-time shadows it does allow baking the lighting and shadows to make the final result look more realistic, as shown in Figure 4.25. Baking basically means pre-computing the shadows of all static objects of the scene and save them as textures.

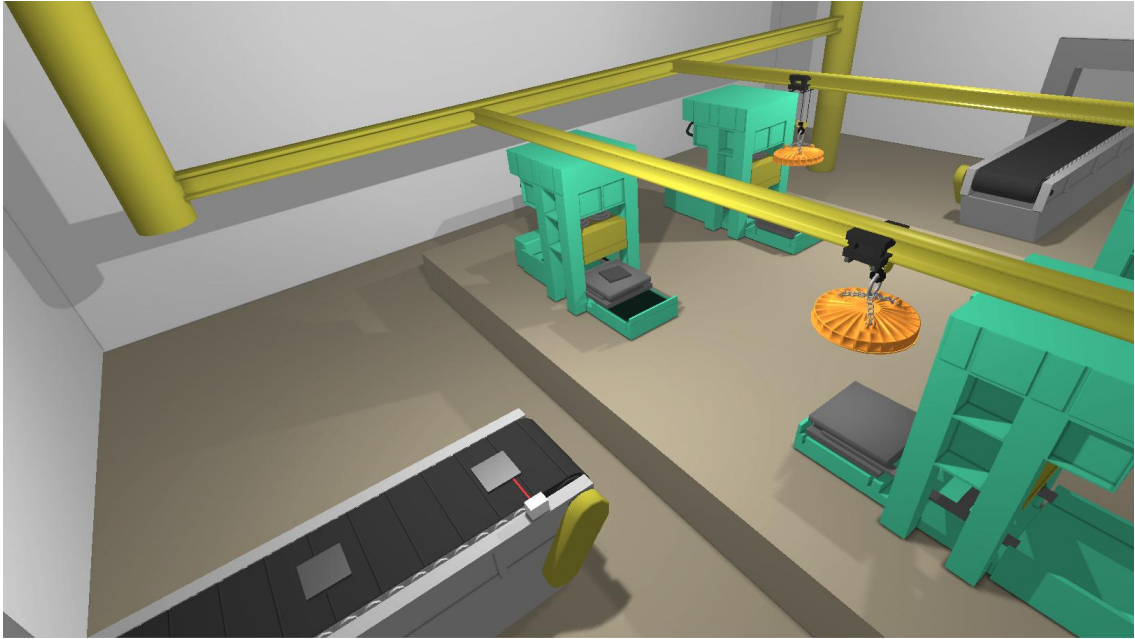


Figure 4.25 Final result of the scene, with 3D models and baked shadows

4.6 Installation

One of the advantages of doing this simulation on Unity is that is really easy to execute on any computer, Unity Webplayer works on Internet Explorer, Firefox, Chrome, Safari and Opera, and it only requires to install a plug-in that needs to be installed the first time the user wants to run any Unity project. In case the user does not have the plug-in the Webplayer will show a link to download it.

4.7 Error simulations

In this section I will explain some of the errors that could happen during the execution of the simulation. Most of them are quite rare but it is worth knowing they can occur.

4.7.1 Physics errors

Using real-time physics creates a lot of potential errors since Unity can skip physics updates to keep the frames per second stable if necessary; this would make the physics unstable if the computer cannot compute them every frame. To avoid this I have decided to slow down the simulation when these issues appear and let the physics engine enough time to compute every physics update.

If the physics have any issue the rest of the simulation may look wrong since the control system does not receive any information about what is really happening so if it thinks it put a piece on a workspace it will grab it again even if the piece fell and is no longer there.

4.7.2 Connection errors

Another source of potential errors is, of course, the connection with the control system, there are many reasons that could make them disconnect, both the client and the server could lost connection at any moment. There is no procedure right now to solve this kind of issues since it is not a major problem, the continuity of the simulation is not that

important, the user does not really lose anything since he is just an spectator watching the simulation.

So, in case of disconnection the simulation stops and shows an error message, giving the user the option to restart the application and reconnect, which reloads the whole scene to restart it.

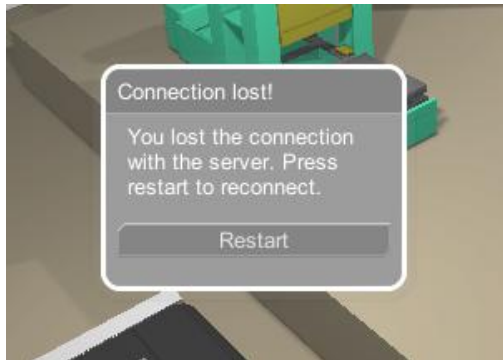


Figure 4.26 Connection lost error message

4.7.3 Control system related errors

Since the control system development is still in progress it can be a source of errors as well, and may command weird actions to the cranes, but these are not going to be discussed in this thesis report, since this thesis only focuses on the simulation.

5 Evaluation

In this section we will check that all goal criteria presented in section 1.2 have been fulfilled.

5.1 Communication in real-time with control software

As explained in sections 3.1 and 4.3 the project uses TCP sockets to reach this criterion, with 2 ports hosted on the control system. Since we are using 2 ports to communicate the server with multiple clients we are using an id system to identify the instances.

This feature has been tested and does work online in real-time.

It has been implemented with the TcpClient class, from .Net.

5.2 Online visualization in browser

The project has been implemented in Unity, so we can build it for Webplayer without encountering many problems and it works on most popular browsers (has been successfully tested on IE9, Firefox and Chrome).

The only comeback is that the first time a computer runs a Unity project it will be asked to download and install the plug-in necessary to run any Unity project, it does not take long and it might already be installed in many computers since Unity is widely used nowadays; however, I can understand that this might be annoying for some users.

Since the control system has been implemented to support multiple instances at the same time many users can see their unique simulation simultaneously.

5.3 Realistic

Since this could be an ambiguous point it needs to be checked with objective arguments:

- Physics simulation: Unity includes a physics engine that properly simulates physics (as good as current physics engine can do it), making the objects fall due to gravity, collide with other bodies and react properly to these collisions.
- 3D models: the models look similar to the researched photo references of the real-life devices (see appendix A to see references of the real objects).
- Lighting: Unity properly simulates the reflection of the light on 3D surfaces using the up-to-date equations.
- Shadows: even though there are no real-time shadows included on the free version of Unity, it allows us to compute the shadows of static objects in the scene (as explained in section 4.5) which is really useful since in that scene most objects can be considered statics; actually, the only objects that do not project shadows are the moving cranes.
- Textures: most of the modeled devices have plain color textures with specular lighting to simulate the metallic surface, but the textures could be more realistic by adding scratches or degradation caused by time and some more details on the diffuse map, and objects could also have had normal maps. This, however, has not been done since spending too much time texturing 3D objects would not have been appropriate for this thesis.

5.4 Interactive

As explained on section 4.4 the simulation includes some GUI controls making it interactive, the controls include:

- Simulation speed: the user can set the simulation speed from 1.0x to 2.0x.

- Cameras: to make it more good-looking and interactive I set 4 cameras in the scene and the user can choose which one he wants to enable and see the scene from it.
- Device control: the user can use slider controls to change the workspaces processing time, the cranes speed and the piece feeding frequency to test a determined control system in different scenarios.

And I also decided to add some statistics about the workspaces work charge, showing the percentage of processed pieces of the total processed pieces and the average inactive time between pieces for each workspace, this does not really bring interactivity, but it gives the user useful data about the control system effectiveness.

6 Conclusion

In this chapter there is a summary of the content of this report, which describes the development of a production cell simulation.

The beginning of the report, chapter 2, discusses the different technologies that were considered to implement this simulation, and discusses which advantages and disadvantages each one has bearing in mind the goal criteria we are looking to fulfill, which are described in section 1.2. The end of the chapter presents the chosen technology, Unity3D, a game engine with a large community of developers that seemed to be the best among the alternatives to help us reach our goal; and also describes which features does Unity already provide and which ones did I need to develop to make the simulation.

Chapter 3 presents the different alternatives regarding the architecture of the system and the communication protocol between the simulation and the existing control system, which is being made by a different person as another thesis. At the end it was decided to have a server-side control system that will be able to control many simulations running on different clients. To keep the amount of sent data low we decided to store the state of the simulation in the control system so the simulation only needs to notify the device changes to the simulation, and the simulation can update its intern state and decide which actions to command to the different devices.

Once the main decisions have been explained, chapter 4 describes the implementation of the simulation. It starts by studying the main Unity features used to make the implementation, including the restrictions of Unity when building for Webplayer, which are really important for the implementation of this project. This chapter continues explaining the implementation of every device in depth.

When the thesis reaches the section 4.3 it presents the implementation of the TCP socket connection, which has been done with .Net's user-friendly interface communication class TcpClient.

The implementation chapter continues explaining the GUI controls, which allow the user to set the properties of the devices in the simulation or choose among different cameras to view the simulation.

In section 4.5 there is a briefly explanation of how the models and lighting of the scene have been made.

Section 4.6 describes how easy is for any user to run the simulation thanks to the Webplayer build made by Unity that works on most web browsers with just a plug-in.

The implementation chapter ends with a list of possible errors the simulation could have and how they are handled.

The last chapter is the evaluation, which checks that all goal criteria have been reached on the final version.

It is important to note that with the final result we achieved our main goal: thanks to this simulation visualization in 3D now we have a way to showcase the control system. This is really useful because without this simulation visualization we would not be able to show how the control system schedules the production cell. At the same time it successfully allows us to test the control system in a 3D environment with physics simulation.

I think the final result includes all the features expected for this project and looks relatively good bearing in mind the modeling and lighting part was not one of the most relevant parts of the project.

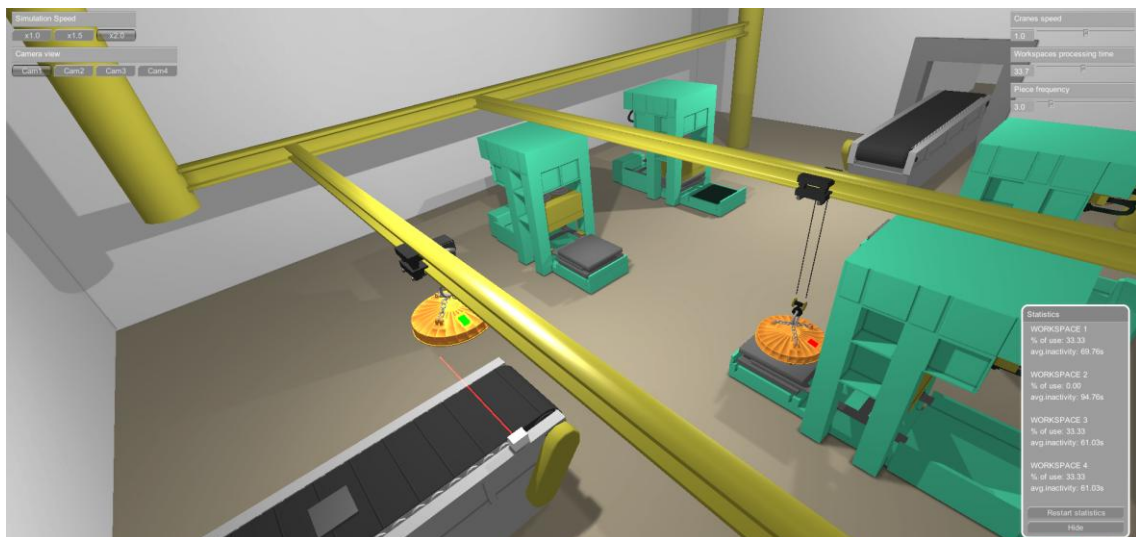


Figure 6.1 Final result of the simulation, from camera 1

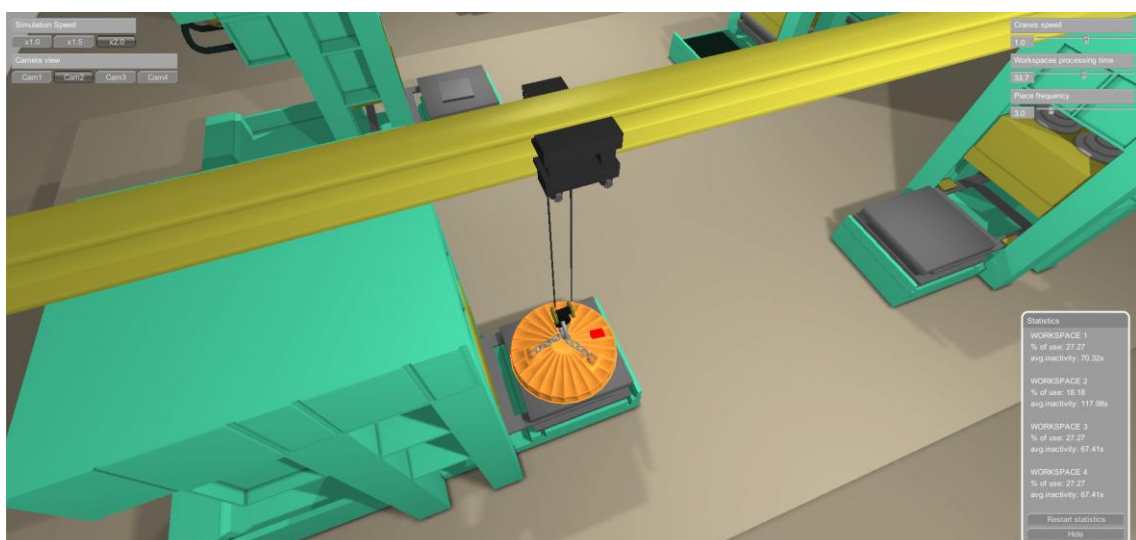


Figure 6.2 Final result of the simulation, from camera 2

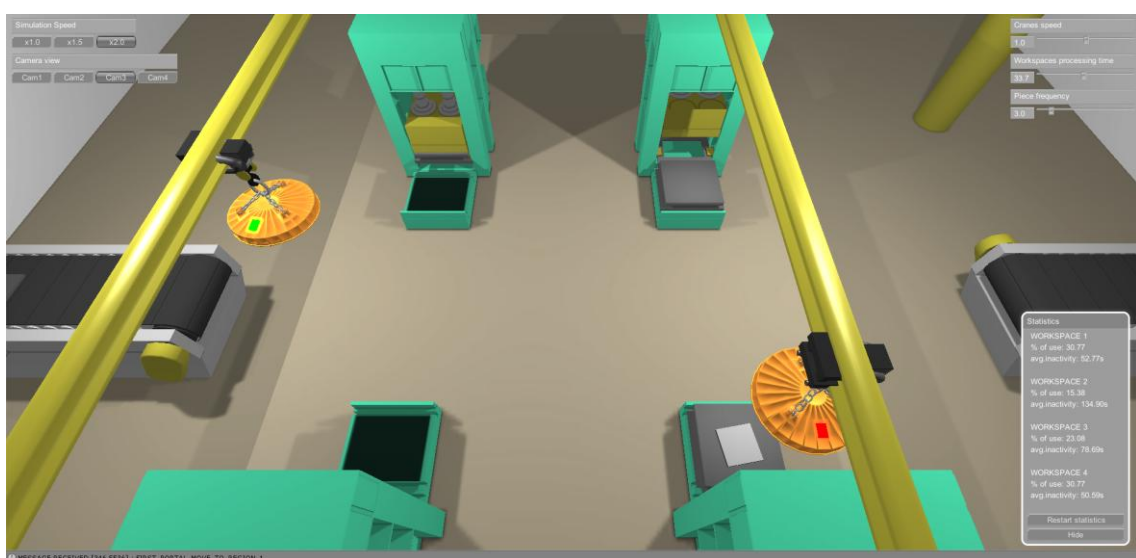


Figure 6.3 Final result of the simulation, from camera 3

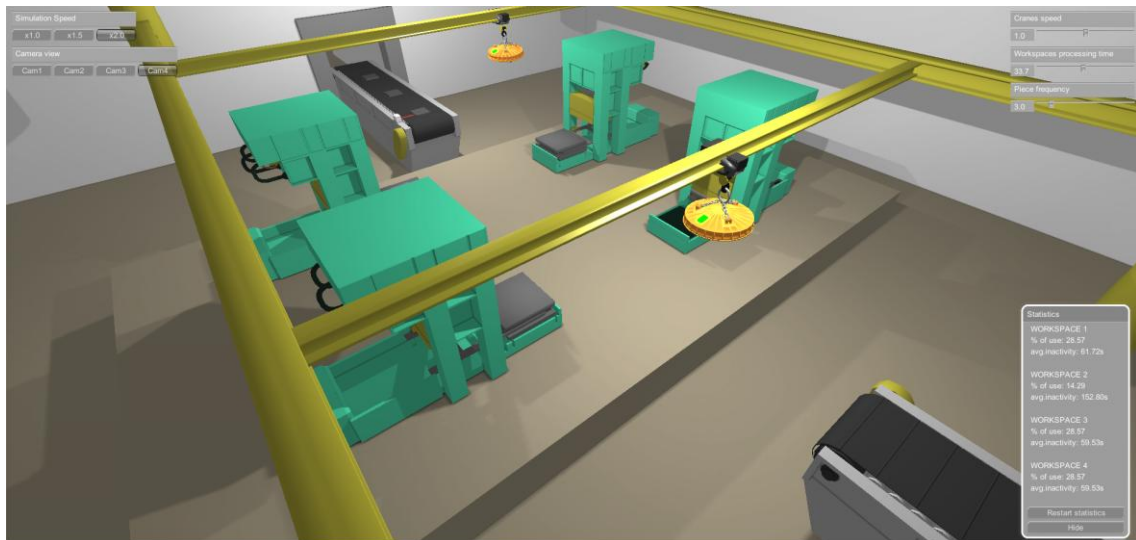


Figure 6.4 Final result of the simulation, from camera 4

6.1 Future Work

As future work the simulation could be improved in some different ways:

- Scalability: even though it would not have been used since the existing control system has been made to control this specific scenario, it could be a great idea to make it scalable and let the control system decide, for example, how many workspaces is it expecting or how many belts, or let it have different types of workspaces.
- Better looks: another improvement could be to have someone work more on the models and illumination, or pay the Unity Pro version to add real-time shadows.
- Error simulation: could also be great to add some error simulation on the devices, like simulate a damaged device that cannot work. And see how the control system solves them.

References

Since game development is a constantly changing field game engines are being updated really often, that makes forums and other web pages the most updated resources in the field and scientific literature or other type of references become out of date very fast. Because of that most of the references used to do this project are web pages.

- [1] *Security Sandbox of the Webplayer*, Unity technologies. Available: <http://docs.unity3d.com/Documentation/Manual/SecuritySandbox.html>. Last accessed (29 June 2013).
- [2] *Indie Games Engines Review*, DevBuzz. Available: <http://www.gamebuzz.me/devbuzz.php/2011/05/20/indie-game-engines-a-recap>. Last accessed (29 June 2013).
- [3] *Unity Webplayer and browser communication*, Unity technologies. Available: <http://docs.unity3d.com/Documentation/Manual/UnityWebPlayerandbrowsercommunication.html>. Last accessed (29 June 2013).
- [4] *Time Manager*, Unity Technologies. Available: <http://docs.unity3d.com/Documentation/Components/class-TimeManager.html>. Last accessed (29 June 2013).
- [5] *What is a Game Engine?*, Game Career Guide. Available: http://www.gamecareerguide.com/features/529/what_is_a_game_.php. Last accessed (29 June 2013).
- [6] *Unity 4 license compare with other game engines*, Unity Community. Available: <http://forum.unity3d.com/threads/141098-Unity-4-license-compare-with-UDK-CRytek-Gamebryo-Unigine-Marmelade-Shiva3D-etc/>. Last accessed (29 June 2013).
- [7] *Unity3D official webpage*. Available: <http://unity3d.com/>. Last accessed (29 June 2013).
- [8] *ShiVa official webpage*. Available: <http://www.stonetrip.com/>. Last accessed (29 June 2013).
- [9] *UDK official webpage*. Available: <http://www.unrealengine.com/udk/>. Last accessed (29 June 2013).
- [10] *CryEngine official webpage*. Available: <http://mycryengine.com/>. Last accessed (29 June 2013).
- [11] *GameKit official forums*. Available: <http://gamekit.org/forum/>. Last accessed (29 June 2013).
- [12] *Delta Engine official webpage*. Available: <http://deltaengine.net/>. Last accessed (29 June 2013).
- [13] *Torque3D official webpage*. Available: <http://www.garagegames.com/products/torque-3d>. Last accessed (29 June 2013).
- [14] *Ogre3D official webpage*. Available: <http://www.ogre3d.org/>. Last accessed (29 June 2013).
- [15] *Unity3D*, Wikipedia. Available: [http://en.wikipedia.org/wiki/Unity_\(game_engine\)](http://en.wikipedia.org/wiki/Unity_(game_engine)) . Last accessed (29 June 2013).
- [16] *Unity official Documentation*, Unity technologies. Available: <http://docs.unity3d.com/Documentation/ScriptReference/index.html>. Last accessed (29 June 2013).
- [17] *TcpClient Class reference*, Microsoft MSDN library. Available: <http://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient.aspx>. Last accessed (29 June 2013).
- [18] A.F.Zorzo, L.A. Cassol, A.L. Nodari, L.A. Oliveira, L.R.Morais , *Long Term Scheduler for Real Time Industrial Installations*

- [19] *Unity fast facts*, Unity technologies. Available:
<http://unity3d.com/company/public-relations>. Last accessed (29 June 2013).
- [20] C. Lewerentz, T. Lindner , *Case study “production cell”: A comparative study in formal specification and verification*, 1995
- [21] C. Lewerentz, T. Lindner , “*Formal Development of Reactive Systems: Case Study Production Cell*”. In *Lectures Notes in Computer Science 891*, Springer-Verlag, 1995
- [22] A. Lötzbeyer, R. Mühlfeld , *Task Description of a Flexible Production Cell with Real Time Properties*, FZI, Karlsruhe, 1996

Appendix A Device references

This appendix shows some references of the real-life devices modeled for the simulation that I researched to make the models, it aims to support the evaluation of the corresponding goal criteria. I do not own any of these pictures.

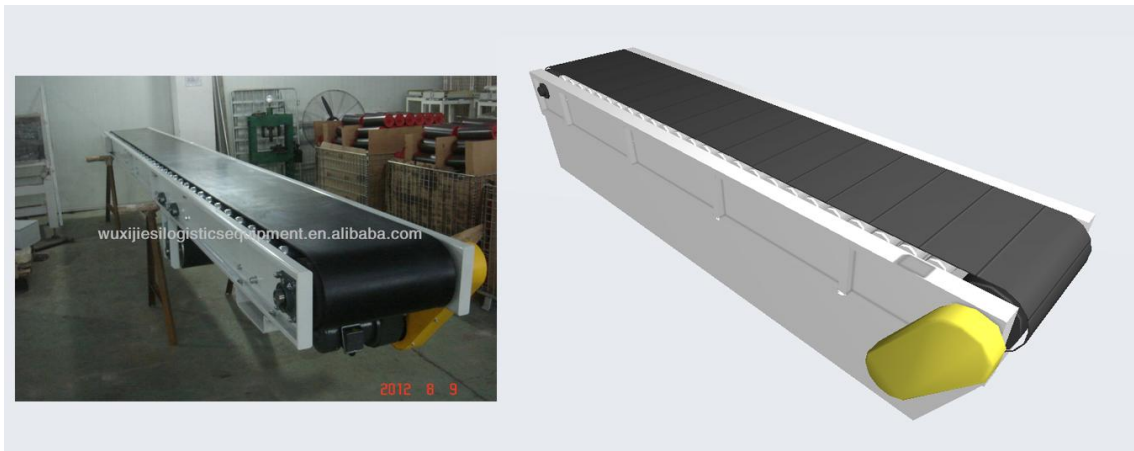


Figure A.1 Belt reference and model

(http://img.alibaba.com/photo/568930529_1/JS_Heavy_duty_belt_conveyor_assembly_line_Material_handling_equipment.jpg)

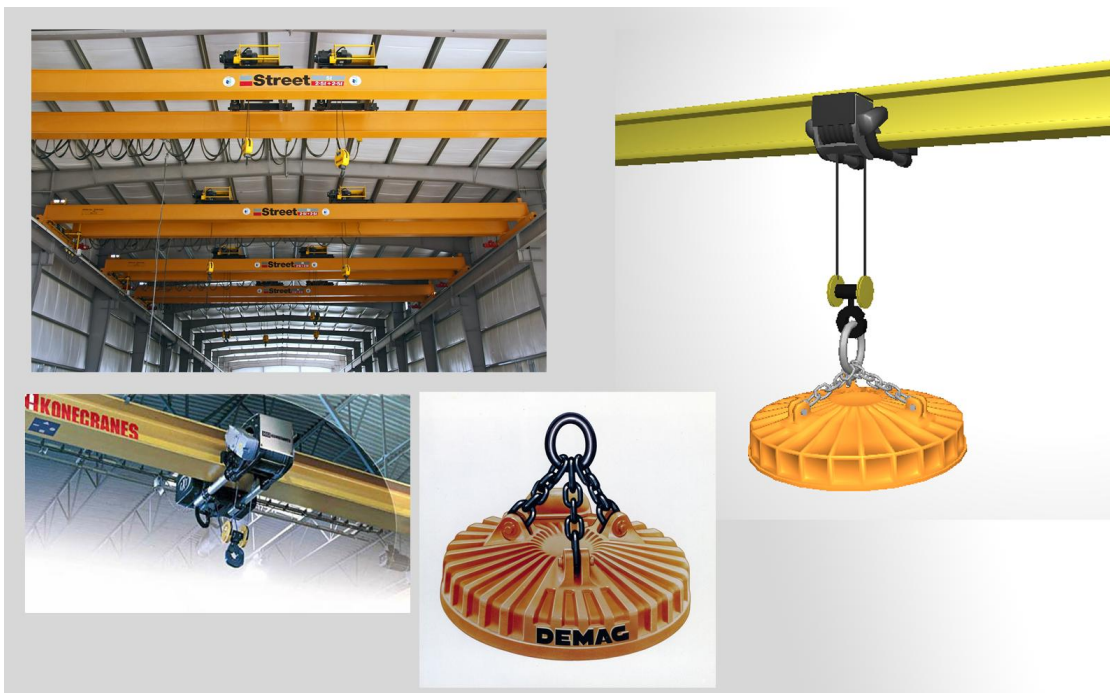


Figure A.2 Crane references and model

(<http://www1.prweb.com/prfiles/2009/11/26/34942/4678st1hjuffalcranes.jpg>, <http://www.prodmoreco-kci.ro/Image002.jpg>, http://img.directindustry.com/images_di/photo-g/circular-lifting-magnet-7651-3440923.jpg)



Figure A.3 Workspace references and model (http://img.directindustry.com/images_di/photo-g/double-action-hydraulic-press-27948-2860323.jpg,
[http://img.alibab.com/photo/319318326/baler for scrap metal.jpg](http://img.alibab.com/photo/319318326/baler%20for%20scrap%20metal.jpg),
http://web.tradekorea.com/upload_file2/product/620/P00250620/cbe9caa5_e81a3a2c_8e17_49cc_aacf_f0a39329cb56.jpg)



Linnæus University

School of Computer Science, Physics and Mathematics

SE-391 82 Kalmar / SE-351 95 Växjö

Tel +46 (0)772-28 80 00

dfm@lnu.se

Lnu.se/dfm