Computer Interface Design Principles

Creating an interactive application

Josef Cohen

2014

Contents

[Task 1: Identifying interactive media systems 2](#_Toc385962488)

[Task 2: Evaluating interface design constrains 3](#_Toc385962489)

[Task 3: Implementing sound effects 4](#_Toc385962490)

[Task 4: Recording and editing sound 5](#_Toc385962491)

[Task 5: Changing object properties on in-game events 6](#_Toc385962492)

[Task 6: The graphical user interface 7](#_Toc385962493)

[Task 7: Special event functions 8](#_Toc385962494)

[Task 8: Game presentation 10](#_Toc385962495)

[Task 9: Game improvements 11](#_Toc385962496)

[Task 10: Additional functionality 11](#_Toc385962497)

[Task 12: Animated explosion effects 11](#_Toc385962498)

[Task 13: Game genre and similarities 12](#_Toc385962499)

[Task 14: Additional improvements 13](#_Toc385962500)

[Task 15: Development activity management 13](#_Toc385962501)

[Task 16: Development team management 13](#_Toc385962502)

[Bibliography 14](#_Toc385962503)

# Task 1: Identifying interactive media systems

Creating interactive media applications has now become a very common thing amongst digital media designers and developers. Perhaps, one of the most popular is the creation of digital games, which are generally developed through what’s famously known as a ‘game engine’.

The most popluar game engines are Unity, UDK and CryEngine. This is mainly because they are the most powerfull game engines yet, and they can be downloaded from the internet for free. Well, you might actually have to pay for an improved version with more features and so.

When deciding which game engine to use, there are some important factors that one needs to consider such as:

**Popularity**: If the game engine is popular amongst a significant ammount of users, then it will progressively have a solid supporting community. This means that more supporting resources and references are to be found around, and the game engine will most likely continue to improve.

**Ease of use**: New users should not be intimidated by the basic workspace, whilst advanced users should simply be able to achieve more in less time, without sacrificing efficiency. There should also be a solid documentation about every aspect of the game engine, with additional video tutorials and sample projects.

**Feature set**: This can be found on the respective game engine’s website. Some popular features include proprietry tools for managing 2D and 3D graphics, editing animation and scripting with various languages. The inclusion of a basic asset set is also important as it may serve usefull for prototyping purposes.

**Integration with third party applications**: Having an extensive set of propriety tools is a good thing. However, there might be situations where it is better to integrate content from a more specialized third party application such as creating custom 3d models with maya, or creating 2D sprites and textures with Photoshop.

**Cross-platform support**: It should provide a quick and easy way to deploy games for several application platforms such as Windows, Android, iOS, etc. There should also be a quick and easy way to optimize the game for various mobile devices and gaming consoles.

<http://unity3d.com/unity>

<https://www.unrealengine.com/products/udk/>

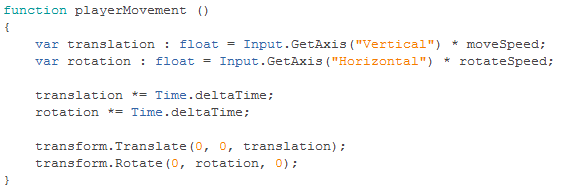
<http://cryengine.com/>

<http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/>

# Task 2: Evaluating interface design constrains

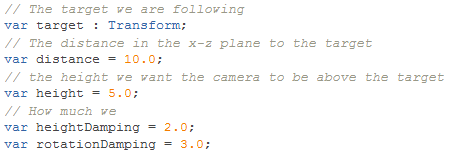
For my personal game project, I created a 3d robot model to act as the player. I wanted it to be controlled in a 3d enviroment, whilst being followed by a third person camera.

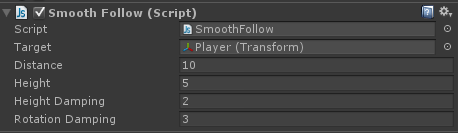
The first thing I scripted was the movement of the player, in which the transform translation and rotation of the player is manipulated through keyboard input.



At this point, the player was able to move, but was eventually disappearing from the scene whenever it was out of the camera’s way. This was obviously due to the fact that the camera wasn’t following the player, so I applied a script to the camera to make it follow the player.

This particular script can be easily found within Unity’s standard scripts package. Basically, it takes a target (the player), and manipulates the camera’s transform properties to follow the player’s transform properties. It’s got some other customizable variables, such as height and distance.





After setting up the script to my personal preferences, the camera was sucessfully following the player, thus fixing the issue of having the player disappear from the scene.

I’ve also experimented with the first person controller found within Unity’s standard package set. However, I wanted my robot model to be fully visible in order to achieve a more interesting look and gameplay experience.

I also decided to work with a fixed screen resolution of 1024 by 768, mainly because I figured that 800 by 600 was a little too small for my needs.



# Task 3: Implementing sound effects

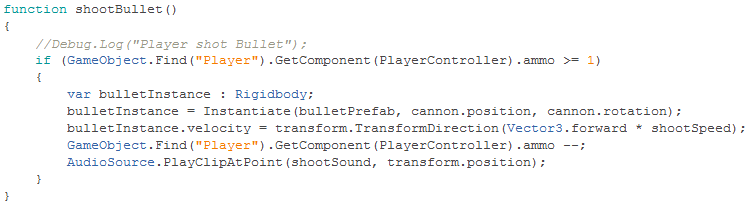
In order to enhance the gameplay experience, I included some sound effects that gets played once throughout certain events. For testing purposes, I downloaded some sounds from the internet, and I imported them into the project’s audio folder.

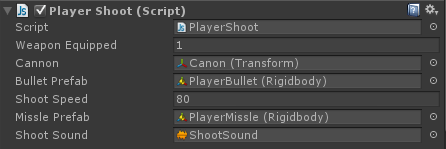
One of the first things I learned was the instantiation of a prefab clone, in which I made the player shoot a bullet with the click of a mouse button. After this was successfully achieved, I put in a function that plays a sound within the ‘if mouse button is clicked’ condition, just after the instantiation process.

In order for the script to work, I had to set up some references which were mainly the AudioSource component to the game object and the AudioClip variable.









The reason for using PlayClipAtPoint instead of some other audio playing function was because of certain situations where I’ve had to delete the game object after instantiation, causing the attached audio source to be deleted too, thus eliminating the reference.

# Task 4: Recording and editing sound

This task is yet to be accomplished…

# Task 5: Object properties and conditions

I wanted to create a situation where if the enemy gets hit by the player’s bullet, the enemy would temporary change its material, making it look like it flashed or something.

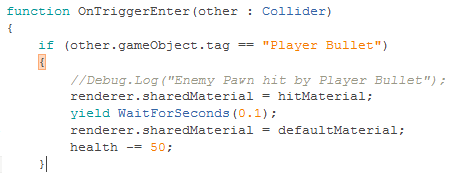
In order to achieve this, I set up both the enemy and the bullet with a collision collider and a rigid body. It was necessary for at least one object to be set as triggers from within its collision settings. This enabled me to use the ‘onTriggerEnter’ function to detect whenever one object’s colliders starts touching another.

I used debug.log to check whether the triggering event was successful or not. Then I set up two variables, one to take the enemy’s default material, and another to take the enemy’s hit material.

I used the ‘player bullet’ tag to specify what happens when the enemy gets in contact with the bullet, rather than what happens when the enemy gets in contact with anthing else.

I accessed the game object’s renderer component to change the material to those provided in the material variables, and I included a short co-routine to quickly switch between materials.







I would have included an in-game screenshot of this event, however it was nearly impossible for me to capture the exact moment of the whole thing happening so fast.

I could also have used the onTriggerExit, however this wasn’t exacly possible since the bullet was destined to be destroyed after getting itself in contact with another object, so I used the co-routine instead.

# Task 6: The graphical user interface

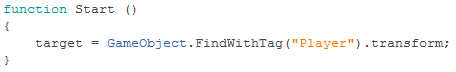
This task is yet to be accomplished…

# Task 7: Special event functions

When scripting in Unity, there are a few ‘special’ event functions that are used to achieve a different kind of result for different situations. The most essential functions are the ‘start’ and the ‘update’, and these two are particularly pre-defined within a newly created script.

**The Start () function:** Anything inside this function starts working immediately when the game loads, more precicely on the frame when a script is loaded during a running application. It does whatever it is made to do in that starting frame and stops, meaning that it doesn’t continue executing code for every frame through the running application. It is commonly used for setting up health, score, etc. values that a character would typically start with on a newly loaded level.

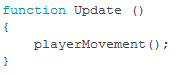
I’ve mainly used it for setting up variables that continually required to contain the same value. For example, I had an enemy prefab that takes the transform properties of the player as the target in order to start following the player. I found myself dragging the player object from the hierchy to the target variable for each enemy prefab I had to put in the scene, so I figured that I needed to have the script find the target by itself from within the start function.

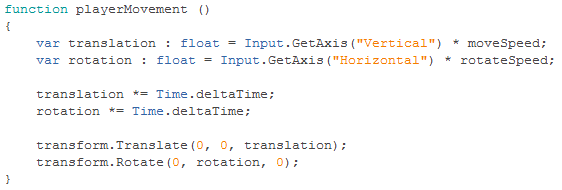


**The Update () function:** Anything inside this function starts working immediately just after the start function is finished from doing its thing. It continues to execute lines of code through every frame of a running application, until the script is no longer available. It is commonly used for camera and character movements, or basically anything that is required to continually change its properties over time, usually within some kind of condition.

When I created the player movement function, I created it separately outside of the update function, then I called it from within the update function. If I hadn’t called it in the update, the player movement function wouldn’t have worked. I could’ve just scripted the whole movement procedure inside the Update function without having to create a separate function, however this may lead to future problems, and may not always work correctly in certain situations.

In the player movement function, the translation and rotation of the player are ready to start continually changing their transform values upon receiving input. However, these values must be multiplied by Time.deltaTime in order to fix the issue of not having a fixed frame rate.

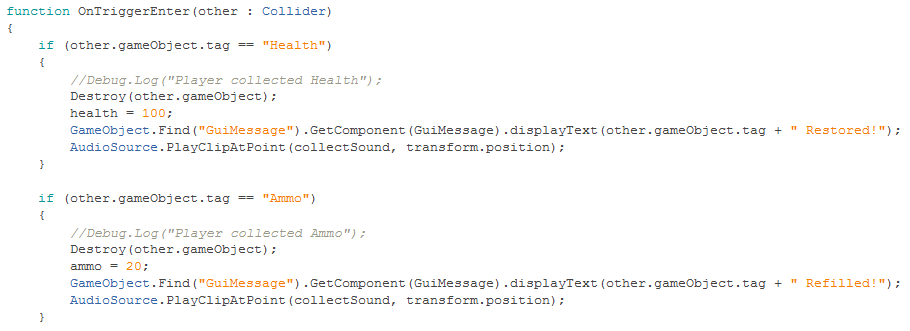




**The OnTriggerEnter () function**: Anything inside this function works whenever an object that is set as a trigger comes in contact with another object, which may also be a trigger. It is commonly used for making the player collect items, or making a bullet apply damage or kill an enemy upon impact. This function isn’t commonly used with static objects that wouldn’t normally react on collision events such as walls and the floor, unless you are going to make such objects trigger some kind of event.

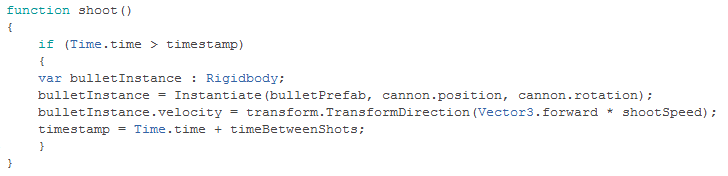
The first time I used this function was when I needed to create three different items which the player could pick. I started experimented with the similar onCollisionEnter function, but then I learned that such functions are commonly used for physics based collision, so I decided not to overcomplicate things for myself before actually getting a solid foundation of the basics.

Basically, this function takes two parameters, the first one (other) is a reference to any other game object that can get hit, and the second one (Collider) is a reference to the game object’s trigger collider itself. I also used tags, which helped me identify certain objects, in order to specify what happens whenever the player gets in contact with a particular type of object, rather than with any object.



I must mention the fact that unlike in the update function, I had no problems implementing co-routines in this function. However, after trying all sorts of things, I did manage to find a solution for making a co-routine work correctly in the update function.





# Task 8: Game presentation

The game ‘Neometrica’ presents ‘the robot’, a character that is ported into a digital world that belongs to some other unfriendly robots.

When you run the executable file, the game will start, and you will be presented with a start menu where you can simply press the space bar on your keyboard to start the first level of the game. You may also press the escape key on your keyboard, if you decide to exit the application.

As soon as the first gameplay level starts, you will be able to start controlling the player immediately through a variety of mouse and keyboard inputs.

|  |  |
| --- | --- |
| Input | Action |
| W key / up arrow | Moves forward |
| S key / down arrow | Moves in reverse |
| A key / left arrow | Rotates to the left |
| D key / right arrow | Rotates to the right |
| Mouse horizontal axis | Look and aim at target |
| Mouse left click | Shoot projectile |
| Alt key | Switch between weapons |
| Esc key | Quit to main menu |

Throughout each levels, you will encouter a number of enemies which are solely instructed to destroy anything that comes in their way.

Once an enemy detects that you are in its presence, it will start following you. It will also start shooting at you, and you will take a small amount of damage from each hit. When you health counter is low, you must quickly find and collect a health pack to fully restore your health.

When the enemy get really close, it will attempt to crash itself onto you, causing an impact that will immediately destroy both you and the enemy. You must also beware of some enemies that may surprise you, so keep an eye out for anything that may come out from suspicios blindspots. Dying would practically mean that you would have to restart the game from the first level, however each level only takes a couple of minutes to complete.

When you see an enemy, you will need to immediately aim and shoot at it until it gets destroyed. With your default weapon is equipped, you can easily take it down with two shots. You must also keep an eye for the ammo counter, or else you’ll end up not being able to shoot. In that case, you will need to quickly find and collect an ammo pack in order to refill you ammo counter. You may also want to switch to a stronger weapon, which shoots a stronger projectile that immediately destroys the enemy with one hit, but it will cost you more ammo.

To complete a level, you must arrive at the end point of the level, where you will find a flag that will take you on to the next level. You will also have a score, which increases with each enemy you destroy, however you can also find and collect some score boosting packs that will additionally increase your score.

After finishing the first three levels of the game, you will encounter the enemy boss, and you must destroy it to successfully complete the game.

# Task 9: Game improvements

If I had more time, I would have tried to use more models, more animations, and more physics. I would also also have tried to achieve a better look for the game, especially for the graphical user interface.

I did spend a fair amount of time trying to figure out how physics work with functions like add force, gravity, etc. However, when things started to get a little bit complicated, I decideded to stick with the basics, since this was my first game development project.

I also didn’t quite fully understand how mechanim works. I knew a few things about it, but the whole blend trees and state machines was something very new to me, so I guess I could have tried to experiment a little bit more with its features.

The game also lacks props, traps, variation of enemies, variation of sounds, cutscenes and a few other things. However, if I were implement all this stuff, I would definitely need a team.

# Task 10: Additional functionality

This task is yet to be accomplished…

# Task 12: Animated explosion effects

This task is yet to be accomplished…

# Task 13: Game genre and similarities

The game ‘Neometrica’ is a shoot ‘em up type of game, in which the player can move around, aim, shoot the enemy, or dodge the enemy’s shots. The player may also collect power ups, and switch between different weapons.

The shoot ‘em up type of game is practically a subgenre of the action genre. The action genre is usually associated with physical challenges, fast decisions and hand to eye coordination. On the contrary to a stealth type game, the player must take on the enemy in a very direct and aggressive manner.

Such games usually require the player to advance by completing a series of leves, where each new level presents a different theme, and more difficult challenges. The camera orientation may vary, some may implement a first person, a third person or a top down camera which follows the player around.

The origins of shoot ‘em up games date back to the late 1970s. Space Invaders is known to be one of the first games such type that was developed during that time, it later inspired the development of games like Asteroids, Galaxian, and Galaga, which were also a massive hit.

Classic games are great, but they are not so pleasently refined, so I had to find a more modern looking game to study its mechanics. After I did bit of research, I found this game ‘Angry Bots’, it was a free sample project by Unity, and it helped me understand better how these kind of games work.

The design of my game was mainly influenced by the popular game ‘Portal’ and the popular film ‘Tron’, where I also make extensive use of robots, neon grids, and the representation of a digital computer world.

# Task 14: Additional improvements

There are a few things about the game ‘Angry Bots’ that I really liked, and I will definitely try my best to implement such features to my game so that it would improve a little more.

The first thing I noticed was the camera movement, it follows the player’s movement, but not the rotation, and I think it would be better for my game if the camera work like this.

I also liked the flash that comes out of the nozzle when the player shoots, and also the laser beacon that helps the player aim and shoot more accurately.

I may also need to work on a pause menu, since my game only includes a start menu and a game over menu, and the player would simply go back the start menu whenever he presses the escape key.

# Task 15: Development activity management

This task is yet to be accomplished…

# Task 16: Development team management

It would truly be a great idea if I had my game to be further developed by a professional team.

If I were to have my own company, I would naturally take the role of the executive director, where I would continually overlook every aspect of the project. Then I would employ some key people:

**Programmer**: This guy would be a genious programmer, he would be assigned to write code, whilst also being able to manage all the technical aspects of the project.

**Graphic Designer**: This guy would be a proficient Photoshop user, he would be assigned to create an extensive set of 2D graphics like textures and GUI elements, whilst also being able to create and showcase concept sketches of such content, including level design.

**3D Animator**: This guy would be a proficient Maya user, he would be assigned to create an extensive set of custom 3D animated characters and props, whilst also being able to create and showcase concept sketches of such content, including character design.

**Sound Designer**: This guy would be a proficient Pro Tools user, he would be assigned to create an extenve set of custom sound effects and music soundtracks.

# Bibliography

**There are no sources in the current document.**