Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Project Deckard

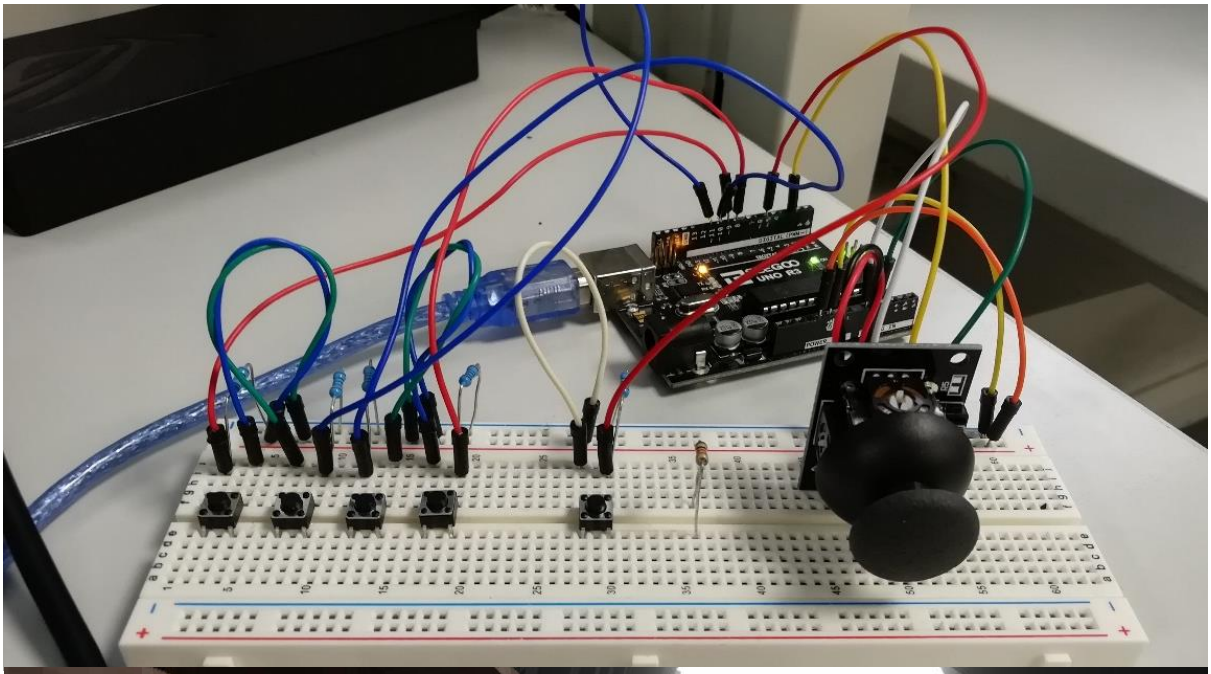
Mághnus O Dómhnaill

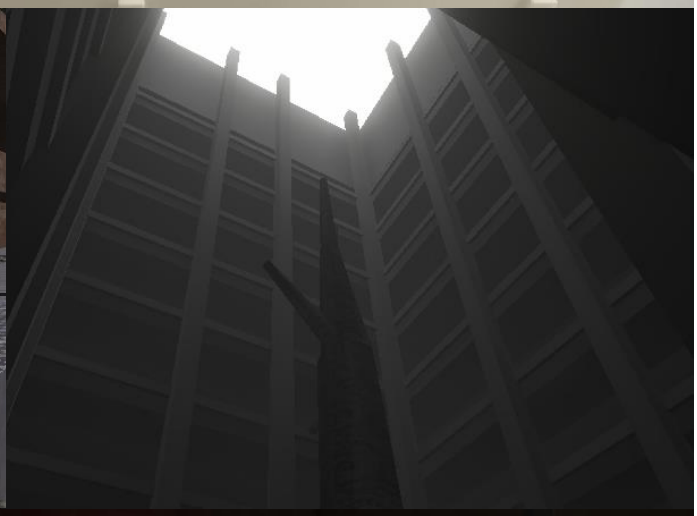
G00377514

Bachelor of Software & Electronic Engineering

Atlantic Technological University

2022/2023






why didn't you say so in the first place, the key's all ready for a new home for 8000 credits
steep enough, anything i can do to take it off your hands
fine, ill take it
you know what, nevermind [EXIT]

Acknowledgements

I would like to thank my project supervisor, Michelle Lynch, for providing me guidance with my project.

I would also like to thank my parents for their support and putting up with me only appearing outside my room for dinner.

Table of Contents

1 Introduction

1.1 Introduction

For my 4th year project, I wanted to make a video game based on some of my favourite media, from films like *Blade Runner* and *Ghost in the Shell*, to video games like *Half-life 2*, *Prey (2017)* and *Deus Ex*. A common theme across the games I enjoy the most is giving the player as much freedom as they can, while allowing restrictions make the player think of out-of-the-box solutions to puzzles created by the game's open systems. Since this is too much for a solo developer, yet alone to an amateur game developer, these systems won't be included. Instead, player choice will be provided by level design and dialogue options with character strewed across the level.

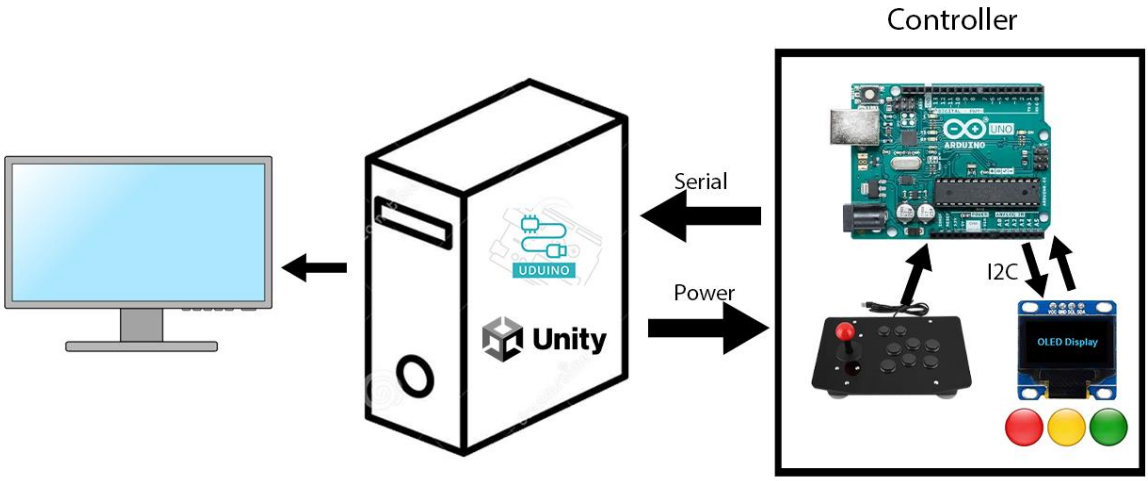
A basic overview of the game is that you'll play as a bounty hunter, tasked by an associate to try and track down and interrogate a suspected fake human. The player will have to talk to other characters on how to gain access to the towering structure the suspect is located in. Once the player has found the suspect, they must interrogate them to find out if they're a real or fake human, with the suspect's true identity chosen every time you start the game. The dialogue of some characters should reflect this change in their dialogue.

1.2 Project Goals

The goals for Project Deckard were:

3 Develop a game using elements of Immersive Simulation design elements.
4 Create a system to allow branching dialogue sequences.
5 Create a gimmick for the game where player must interrogate a group of suspects.
6 Develop a controller using an Arduino and a selection of buttons.
7 Integrate the controller into the game with equivalent functionality to game console controllers.

2 Block Diagram



3 Body of the Project

3.1 Unity

Unity is a free cross-platform game engine developed by *Unity Technologies*, announced and released at the Apple Worldwide Developers Conference in June 2005. It allows developers to create games in both 2D and 3D forms and in recent years seen more adoption in other industries such as virtual reality, film, architecture, construction, and limited use by the American Armed Forces (Wikipedia, 2023). Due to its ability to run on mobile devices, many game developers use Unity for the fast-growing mobile games market.

Unity is widely accepted as the best way to learn game development, as its programming language is C#. It's competitors *Unreal Engine* and *Godot* both use C++ and while Unreal Engine allows its users to use Visual Scripting, its often frowned upon if the user relies solely on it. Hence Unity is recommended as it teaches its user better coding etiquette and in turn, Unity has more robust documentation for beginners, making it perfect to begin quickly developing a prototype.

3.2 Blender

Blender is a free, open source 3d modelling and animation program maintained by the non-profit Blender Foundation. It allows users to create models, animate, and simulate fluids and other materials. Due to it being free, its seen wide adoption in many industries and communities, especially game development by small game development studios and individuals. Much like Unity, its documentation is quite robust and there are many tutorials on websites like YouTube to learn from.

3.3 Arduino

The Arduino is an open source, hardware and software development platform, developed by a group of college students as a master's project for allowing non-engineers to develop their own devices. Thanks to its large community, finding plugins to allow communication between an Arduino and the Unity Editor was very easy. The two main plugins were *Uduino* and *Ardity*. Ardity is a free plugin but initial testing with a dummy project proved it unreliable. Uduino, while costing €15, was significantly more reliable and stable than its free alternative and thus perfect for the controller.

3.4 C# Programming Language

A C-based, general purpose, high-level object-oriented programming language developed by Microsoft in 2000. C# has many similarities to other programming languages like Java and C++, meaning it'll be very easy to pick up on its nuances.

4 Dialogue Interactions

Since the player is playing the game as a bounty hunter, asking characters in the game for information on the suspects is very important, adding to the puzzle of trying to find said suspect. To achieve this, a system of displaying dialogue at runtime was necessary. This would later be modified to allow the main gimmick of the game, interrogating suspects to figure out if they're a real or fake human, to function properly.

When developing the dialogue system, the initial version stored the dialogue in an array and to cycle through it, a button on the screen would need to be pressed and a switch case would then display the new dialogue.

```
Unity Message | 0 references
void Update()
{
    ray = camera.ScreenPointToRay(Input.mousePosition);

    if (Physics.Raycast(ray, out hit, ray_length) && Input.GetButtonDown("Fire1"))
    {

        switch (hit.collider.gameObject.name)
        {
            case "button1":
                TriggerDialogue(i);
                buttons_pressed++;
                i += 3;

                break;
            case "button2":
                TriggerDialogue(i + 1);
                aggression += 1;
                buttons_pressed++;
                i += 3;

                break;
            case "button3":
                TriggerDialogue(i + 2);
                aggression += 2;
                buttons_pressed++;
                i += 3;

                break;
            default:
                Debug.Log("Raycast hit object: " + hit.collider.gameObject.name);
                break;
        }
```

Figure 4.1 - Original iteration of dialogue system

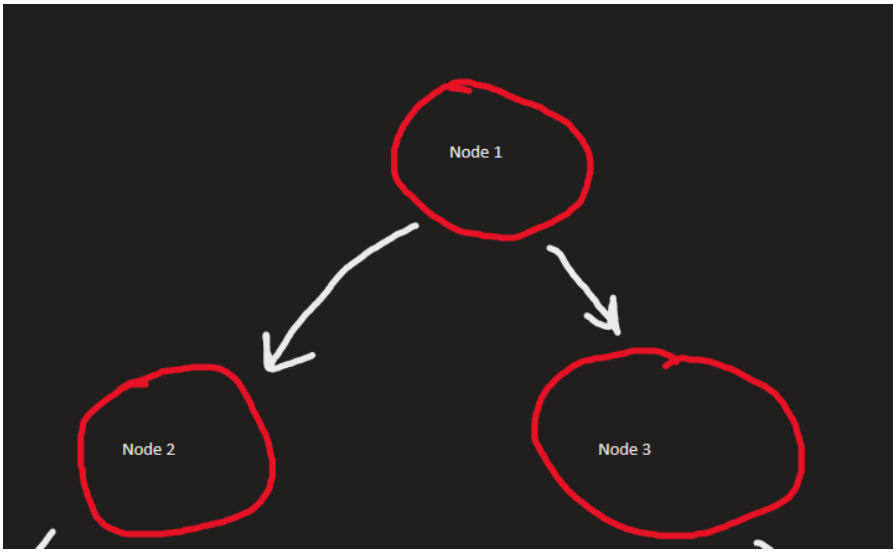
While this did work in the beginning, it was very prone to bugs. There had to be three sentences of dialogue or else the script would crash, the dialogue had to be hard coded into the script itself, and one of the project goals of creating branching dialogue was very difficult to achieve using this method. To achieve the desired results, I needed to store the dialogue data in another file called a *Scriptable Object*.

4.1 Dialogue Scriptable Object

There are multiple ways for storing data in Unity, such as storing data in text documents in a JSON format. This is commonly used to store video game settings like game volume, resolution, graphics settings, and key binds for controls. While a format like this is useful, in the case of storing dialogue, it's a bit overkill especially for a student project. The second and more useful format is in the form of a scriptable object. Scriptable Objects are a data container that can store vast amounts of data, independent of class instances (Unity, 2021). Here is an example of a scriptable object.

The main advantage of a scriptable object is that it allows us to create an "asset" that stores data, similar to other assets like models, sound effects, and materials. These assets are then accessed by game assets within the current Unity Scene. When creating a game asset, its good practice to create a "prefab" of it. This prefab is the master copy, meaning any change done to the main file will also transfer to the copies in the scene, but changes to the copies don't affect the master copy. The only drawback is that each copy has its own copy of data, in turn consuming some memory. To reduce the impact, the prefabs reference data from a scriptable object and use that instead. By containing the dialogue in a scriptable object, multiple characters can use it seamlessly.

The next step was to figure out a way of creating a dialogue system that allowed for easy scalability. The best way of doing this was to have the dialogue use a binary tree format.



In a binary tree, a "node" will have at max 2 child nodes. In the diagram above, the child nodes of node 1 are "2" and "3". By using a format like this, the branching dialogue can be scaled very easy. In our case, instead of having two child nodes, we'll have three. This lets the player have at max 3 options for more dialogue, or 2 options and an exit option.

To achieve the binary tree design, we'll need to create multiple classes, consisting of *DialogueTreeScriptableObject*, *DialogueBranch*, *DialogueSection*, and *DialogueResponse*.

- The *DialogueBranch* class will contain a string called *branch_name*, a branch ID, and a *DialogueSection* array.
- The *DialogueSection* class contains a string called *dialogue* which will contain the dialogue of the character the player will interact with, and a *DialogueResponse* array.
- The *DialogueResponse* class contains several Booleans, *next_branch_id* for what branch to display next, and a string containing the response of the player character. *End_on_response* lets the *Dialogue Manager* know that when we press this response to close the dialogue menu. *Initialize_interrogation* will begin the main gimmick of the game of interrogating the suspects. *Turn_hostile*, as the name implies, will turn the character we're talking to hostile and attack the player. Finally, *event_trigger*, which will allow other scripts to execute functions if the response is pressed, such as giving the player a new item.

```
public class DialogueTreeScriptableObject : ScriptableObject
{
    public DialogueBranch[] branches;
}

[System.Serializable]
3 references
public class DialogueBranch
{
    public string branch_name;
    public int branch_id;

    public DialogueSection[] sections;
}

[System.Serializable]
1 reference
public class DialogueSection
{
    [TextArea]
    public string dialogue;

    public DialogueResponse[] responses;
}

[System.Serializable]
1 reference
public class DialogueResponse
{
    //  ensures player can exit dialogue when conversation is completed
    public bool end_on_response = false;
    public bool initialize_interrogation = false;
    public bool turn_hostile = false;
    public bool event_trigger = false;

    public int next_branch_id;

    [TextArea]
    public string response_dialogue;
}
```

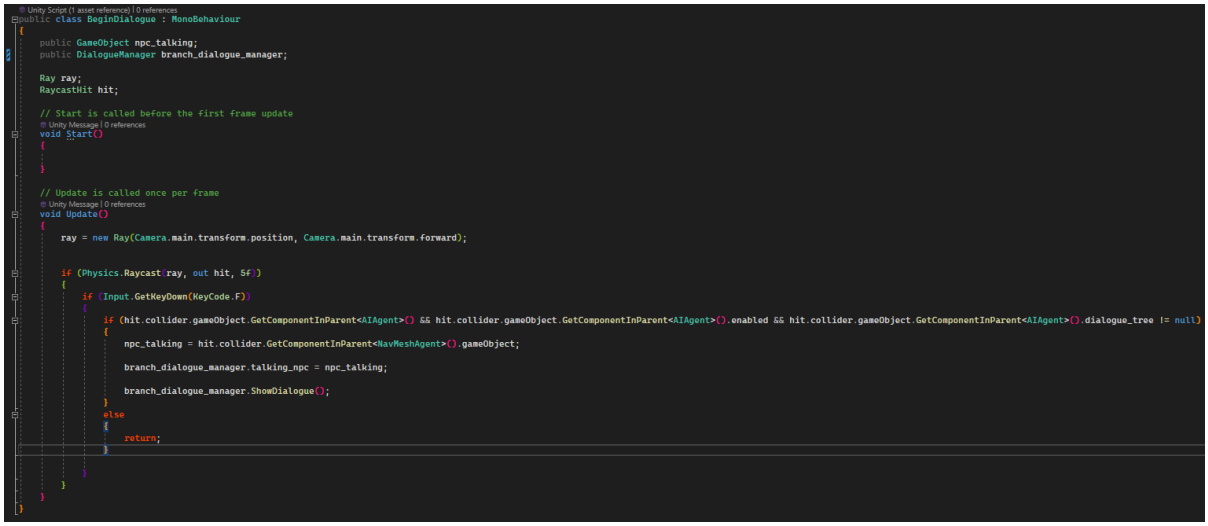
With the Dialogue tree scriptable object finished, we'll need to display it using another script called *Dialogue Manager*.

4.2 Dialogue Manager

In Unity, there are scripts called “Managers”. These scripts centralize the logic for game objects, for instance, any User Interface (UI) elements that is accessed by the player or Non-Playable Characters (NPCs), they access it via a manager class called *UIManager*. This keeps much of the code neat, centralized and easier to read and trace.

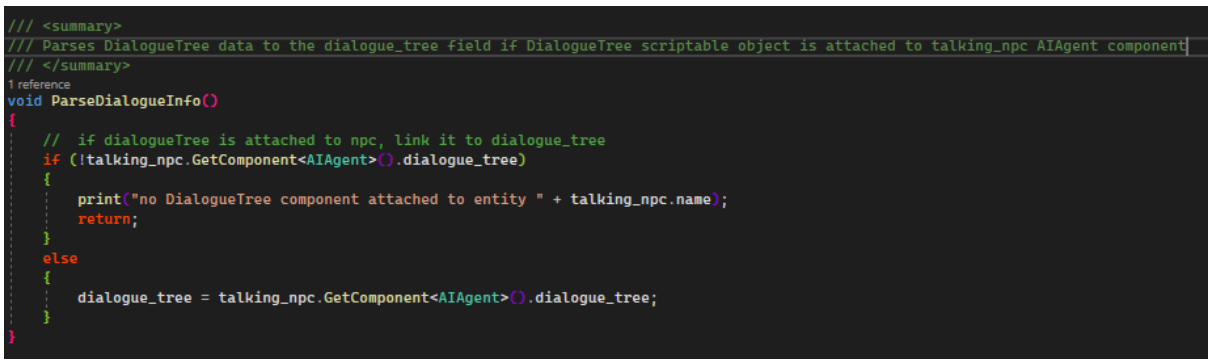
For the Dialogue Manager, we need it to receive the dialogue of the Non-Playable Character, NPC, the player is talking to, display said dialogue, read what responses the player has picked, and respond accordingly to the Booleans in the *DialogueResponse* class.

To begin the dialogue sequence, the player must find an NPC with a *DialogueTreeScriptableObject* referenced in their *AIAgent* script. When pressing the key “F”, a ray is cast from the centre of the camera outwards by 5 units. If the ray hit an object that contained both the *AIAgent* script and the referenced *DialogueTreeScriptableObject*, it will store the character in a public GameObject field.



```
Unity Script (1 asset reference) | 0 references
public class BeginDialogue : MonoBehaviour
{
    public GameObject npc_talking;
    public DialogueManager branch_dialogue_manager;

    Ray ray;
    RaycastHit hit;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        ray = new Ray(Camera.main.transform.position, Camera.main.transform.forward);


        if (Physics.Raycast(ray, out hit, 5f))
        {
            if (Input.GetKeyDown(KeyCode.F))
            {
                if (hit.collider.gameObject.GetComponentInParent<AIAgent>() && hit.collider.gameObject.GetComponentInParent<AIAgent>().enabled && hit.collider.gameObject.GetComponentInParent<AIAgent>().dialogue_tree != null)
                {
                    npc_talking = hit.collider.GetComponentInParent<NavMeshAgent>().gameObject;

                    branch_dialogue_manager.talking_npc = npc_talking;

                    branch_dialogue_manager.ShowDialogue();
                }
                else
                {
                    return;
                }
            }
        }
    }
}
```

The NPC is then referenced in the *DialogueManager* class to ensure we only receive the data stored in this specific NPC. The Dialogue Manager then checks if the NPC has the dialogue scriptable object and if so, parses the *DialogueTreeScriptableObject* scriptable object to the field *dialogue_tree*.



```
/// <summary>
/// Parses DialogueTree data to the dialogue_tree field if DialogueTree scriptable object is attached to talking_npc AIAgent component
/// </summary>
1 reference
void ParseDialogueInfo()
{
    //  if dialogueTree is attached to npc, link it to dialogue_tree
    if (!talking_npc.GetComponent<AIAgent>().dialogue_tree)
    {
        print("no DialogueTree component attached to entity " + talking_npc.name);
        return;
    }
    else
    {
        dialogue_tree = talking_npc.GetComponent<AIAgent>().dialogue_tree;
    }
}
```

Next, the dialogue must be displayed on the screen. I made a quick dialogue box with some text and three buttons.

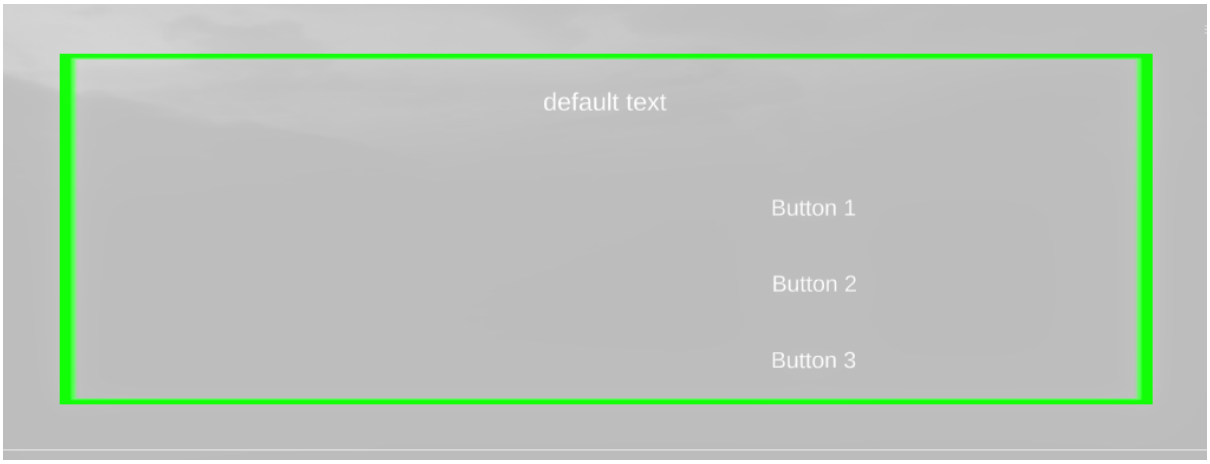


Figure 4.2.1 - Default state of dialogue box

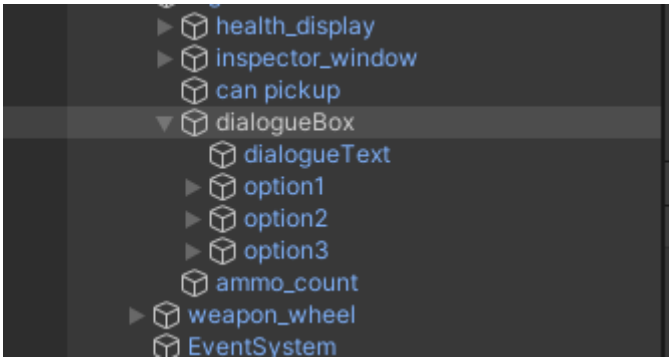


Figure 4.2.2 - Dialogue box model parts.

The default text must be changed to the dialogue string in *DialogueSection* class. Since the dialogue will always start at the beginning, the *branch_id* for the first *DialogueBranch* will be "0" by default. I created a method to quickly check if any of the *branch_id's* had the value of 0 and if so, it will parse the dialogue string to the UI and change the text.

```
2 references
public void DisplayConvo()
{
    //  displays dialogue text in textbox

    foreach(DialogueBranch dialogueBranch in dialogue_tree.branches)
    {
        if (dialogueBranch.branch_id == branch_choice)
        {
            dialogue_branch = dialogueBranch;
            dialogue_text.text = dialogueBranch.sections[0].dialogue;
        }
        else { }


    }

    //dialogue_text.text = dialogue_tree.branches[branch_choice].sections[0].dialogue;

    DisplayResponses();
}
```

Next thing was to check now many responses the player had. We check the length of the responses array and depending on the number of them, we set some of the buttons as active or inactive. If there is only one response, button 1 is enabled, if there are two responses, buttons 1 and 2 are enabled, if there are three responses, all buttons will be enabled and displayed.

```
/// <summary>
/// Reads number of dialogue responses and sets appropriate buttons as active
/// </summary>
1 reference
void DisplayResponses()
{
    //  checks number of responses, turns off buttons depending on size of dialogue responses
    int responses_count = dialogue_branch.sections[0].responses.GetLength(0);

    //  scitch case to vary amout of buttons displayed on display, along with configuring them for interaction
    switch (responses_count)
    {
        case 1:
            //  only sets button1 as active. rest are set is inactive
            button1.SetActive(true);
            button2.SetActive(false);
            button3.SetActive(false);

            //  configures button1 to display dialogue responses
            ConfigureButton(button1);
            break;

        case 2:
            //  only sets button1 and button2 as active. button3 is set as inactive
            button1.SetActive(true);
            button2.SetActive(true);
            button3.SetActive(false);

            //  configures button1 and button2 to display dialogue responses
            ConfigureButton(button1);
            ConfigureButton(button2);
            break;

        case 3:
            //  sets all buttons as active
            button1.SetActive(true);
            button2.SetActive(true);
            button3.SetActive(true);

            //  configures all buttons to display appropriate dialogue responses
            ConfigureButton(button1);
            ConfigureButton(button2);
            ConfigureButton(button3);
            break;

        default:
            break;

    }

    //  resets response_choice for next node of responses
    response_choice = 0;
}
```

To configure the buttons, we create fields to reference them at the top of the script. The *ConfigureButton* method will change the text elements attached to the buttons, as well as change the *choice* and *branch_choice* fields of the script *DialogueBranchButton* to the values of *response_choice* and *next_branch_id* contained in the *DialogueTreeScriptableObject.*

```
    public GameObject button1 = null, button2 = null, button3 = null;       //  pressable buttons for dialogue
    public GameObject vk_button1 = null, vk_button2 = null, vk_button3 = null;      //  interrogation button buttons
```

```
/// <summary>
/// Configures parsed button gameobject with a TextMeshProUGUI component with appropriate dialogue responses
/// </summary>
/// <param name="button">button gameobject to be configured</param>
6 references
void ConfigureButton(GameObject button)
{

    //  button text is parsed the response dialogue
    button.GetComponentInChildren<TextMeshProUGUI>().text = dialogue_branch.sections[0].responses[response_choice].response_dialogue;

    if (!button.GetComponent<DialogueBranchButton>())
    {
        return;
    }
    else
    {
        button.GetComponent<DialogueBranchButton>().choice = response_choice;
        button.GetComponent<DialogueBranchButton>().branch_choice = dialogue_branch.sections[0].responses[response_choice++].next_branch_id;      //  increments response_choice for next button
    }
}
```

The buttons contain a small script that upon left clicking them with the mouse cursor, they execute a method that will send the *branch_choice* and *choice* values back to the Dialogue Manager.

```
public class DialogueBranchButton : MonoBehaviour
{
    public DialogueManager branch_dialogue_manager;


    public bool is_pressed = false;
    public int branch_choice = 1;
    public int choice = 0;

    0 references
    public void OnPress()
    {
        branch_dialogue_manager.UpdateConvo(branch_choice, choice);

        print("Pressed button");
    }
}
```

UpdateConvo will then check which *DialogueResponse* class contains the same *response_choice* value as the pressed button. If the response has either *end_on_response* or *turn_hostile* set as true, the dialogue box will be set is inactive, hiding it from the player's screen until they begin a conversation with another character. If *initialize_interrogation* is set as true, it will begin executing methods used solely for the interrogation mini-game. If neither are enabled, we display the new branch dialogue and responses until the player exits.

```
/// <summary>
/// Updates the dialogue shown on screen
/// </summary>
/// <param name="branch_choice">Integer assigned to button parse from dialogue_tree's next_branch_id parameter</param>
/// <param name="response_choice">Integer parsed from button script to chose respose to dialogue, 3 buttons = integer values of 0 -> 2</param>
1 reference
public void UpdateConvo(int branch_choice, int response_choice)
{
    if (dialogue_branch.sections[0].responses[response_choice].end_on_response)
    {
        print("exited conversation");
        HideDialogue();
        in_convo = false;
    } else if (dialogue_branch.sections[0].responses[response_choice].initialize_interrogation)
    {
        print("begun interrogation");
        //HideDialogue();
        kit.SetActive(true);
        kit.GetComponent<VKKit>().kit_cam.enabled = true;
        in_convo = true;
        DisplayInterrogation();
    }
    else if (dialogue_branch.sections[0].responses[response_choice].turn_hostile)
    {
        talking_npc.GetComponent<AIAgent>().stateMachine.ChangeState(AIStateID.AttackPlayer);
        HideDialogue();
    }
    else
    {
        this.branch_choice = branch_choice;
        DisplayConvo();
        print("continued conversation");
    }
}
```

The end results are displayed below, along with the *DialogueTreeScriptableObject* data:

The first branch's dialogue and responses are displayed on the screen.

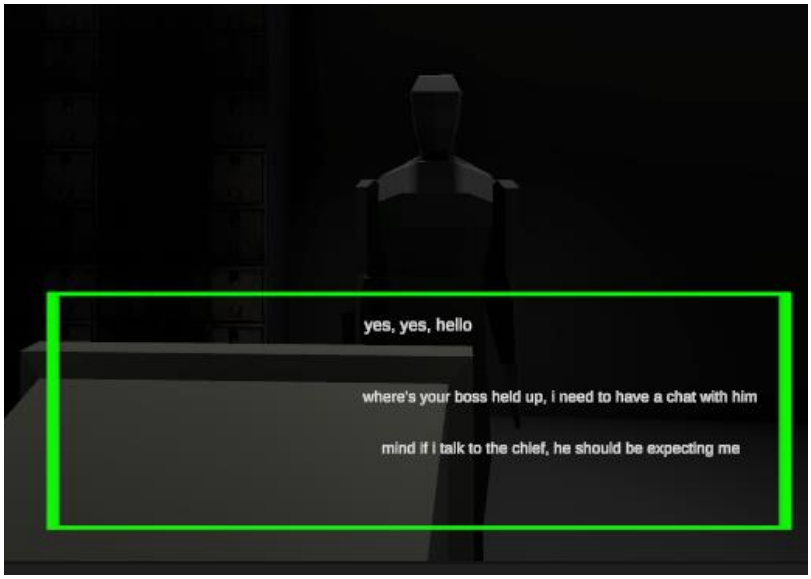


Figure 3.2.3 - Non-Playable Character in Police Station that the player can talk to.

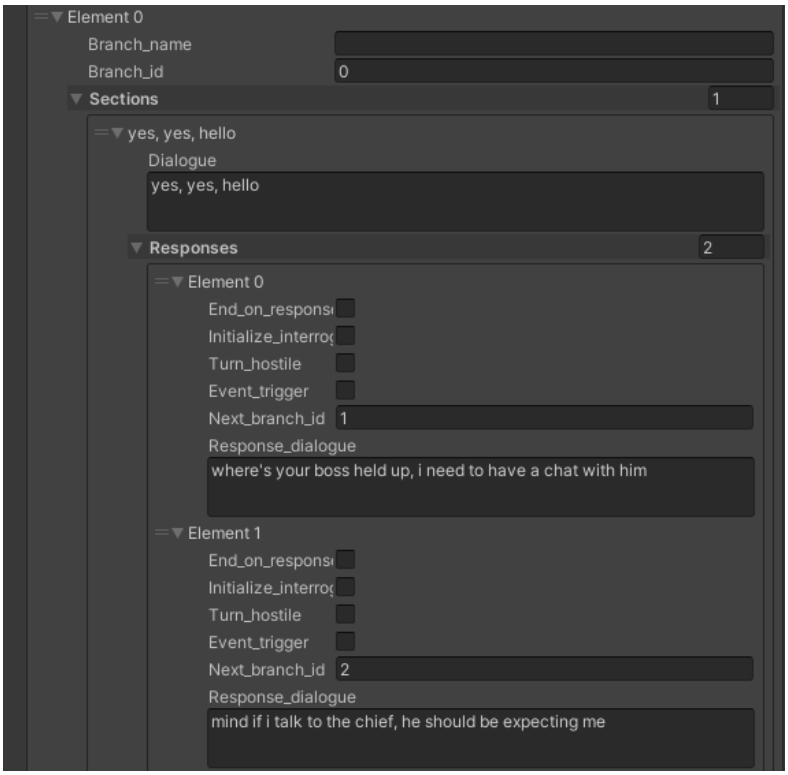
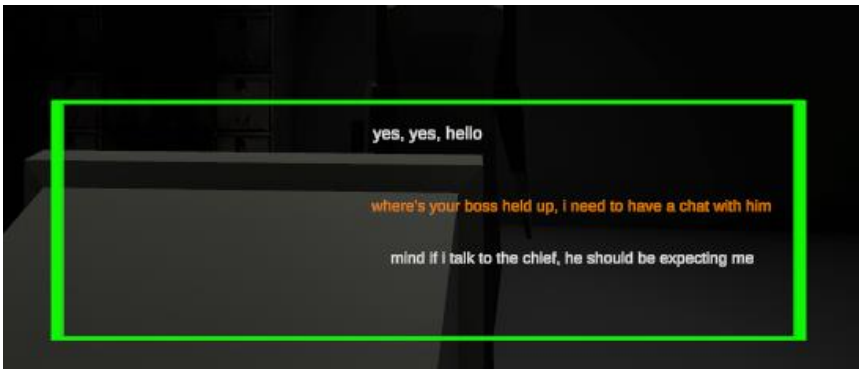
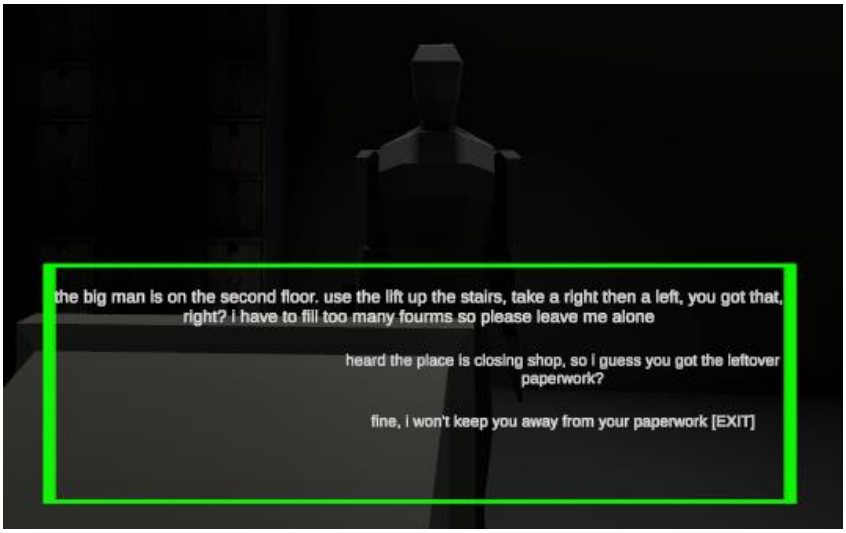


Figure 4.2.4 - Dialogue displayed above in DialogueTreeScriptableObject.

Hovering the mouse over the responses changes the colour, indicating that it's a pressable button.



When the response is pressed, the dialogue and responses are changed at runtime.



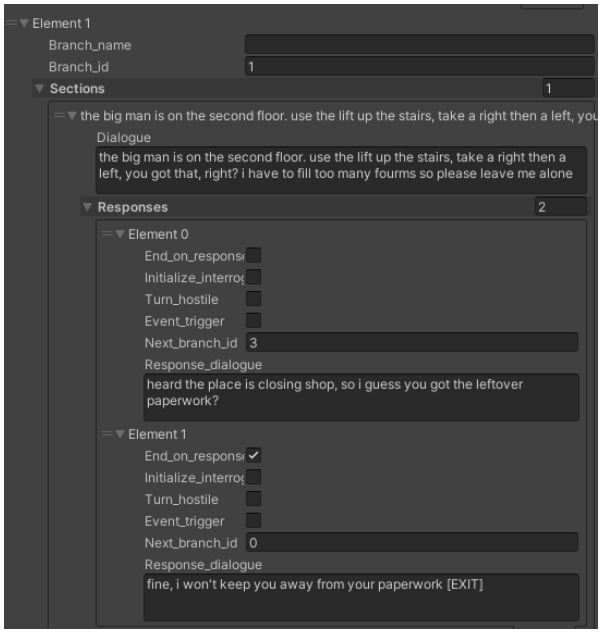


Figure 4.2.5 - New displayed dialogue and scriptable object data.

5 Interrogation Mini-game

To create the interrogation mini-game, a few rules must be set. There must be three or four questions, with three ways of saying them; relaxed, neutral, and aggressive. For example, if questioning the suspect on where they were, the questions would be as follows:

- Relaxed: Did you go out on Saturday?
- Neutral: Where you around this area at 10.21pm on Saturday night?
- Aggressive: You did attack this person on Saturday night, didn't you.

The player must also experience consequences for picking a question. A relaxed question might not give enough information to be useful, a neutral question would give a better idea but still not have enough conclusive information, and an aggressive question might give the clearest answer but would provoke the suspect into attacking the player if picked too often.

If the player were unable to find enough information on whether the suspect was the target, notes or clues must me left around the area. To keep up with the theme of player freedom, there must also me options to give to the player on if they want to eliminate the target or let them go free.

5.1 Interrogation Dialogue Scriptable Object

The scriptable object for the interrogation dialogue will have an identical structure to the dialogue scriptable mentioned previously, using the format of a binary tree, as seen below.

```
public class InterrogationDialogueTreeScriptableObject : ScriptableObject
{
    public InterrogationDialogueBranch[] branches;

    public bool is_real_human = true;
    public DialogueTreeScriptableObject post_interrogation_dialogue;

    public float reveal_human_type_level = 4;
    public float turn_hostile_level = 8;
}

[System.Serializable]
1 reference
public class InterrogationDialogueBranch
{
    public string branch_name;
    public int branch_id;
    //public bool end_on_final;

    public InterrogationDialogueSection[] sections;
}

[System.Serializable]
1 reference
public class InterrogationDialogueSection
{
    [TextArea]
    public string dialogue;

    public InterrogationDialogueResponse[] responses;
}

[System.Serializable]
1 reference
public class InterrogationDialogueResponse
{
    //  ensures player can exit dialogue when conversation is completed
    public bool end_on_response = false;
    public bool turn_hostile = false;
    public float add_aggression;


    public int next_branch_id;

    [TextArea]
    public string response_dialogue;
}
```

The Interrogation Dialogue Scriptable Object is very similar to the previously mentioned Dialogue Scriptable Object, with some minor additions. The main file contains a field to reference another Dialogue Scriptable Object if we want new dialogue to occur after the interrogation mini game. There is also a Boolean for if the suspect is a real or fake, a float called *reveal_human_type_level* that the player must exceed for the game to reveal if the suspect is the target, and another float called *turn_hostile_level* that if the player exceeds, it will make the suspect aggressive and attack the player.

The *InterrogationResponse* class is mostly unchanged but has an additional float called *add_aggression* that upon pressing the appropriate button will add this value to the aggression level of the suspect.

5.2 Interrogation mini-game functionality

The Interrogation mini game begins much like the dialogue, we display the dialogue text from the first element of the *InterrogationDialogueBranch* class and then display the responses on the buttons.

```
//  Interrogation functions

1 reference
public void DisplayInterrogation()
{
    in_interrogation = true;

    //ShowDialogue();

    dialogue_text.text = interrogation_dialogue_tree.branches[branch_choice].sections[0].dialogue;


    DisplayInterrogationResponses();
}
```

Again, we'll check how many responses the player will have. This is because the exit dialogue contains only one response unlike the usual three. Then, the buttons are configured by parsing the response text to the button text component. The text on the buttons will be displayed on the screen to show what the player can ask, but the buttons themselves will do nothing to progress the interrogation mini game.

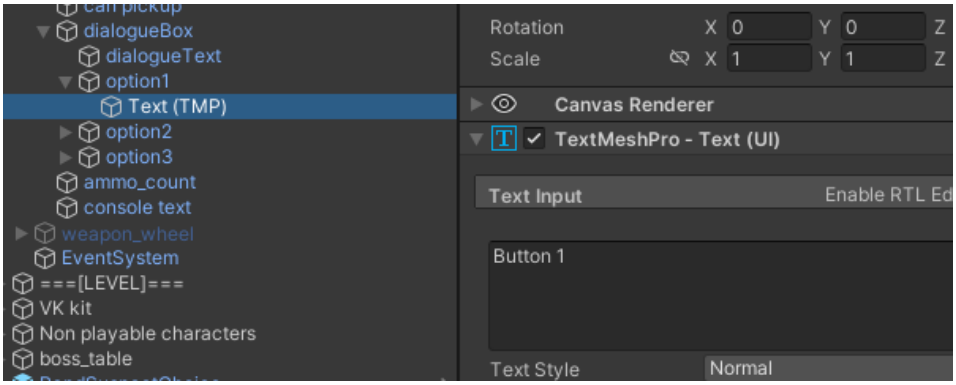


Figure 5.2.1 - Dialogue box game object and TextMeshPro component.

```
2 references
void DisplayInterrogationResponses()
{
    int size = interrogation_dialogue_tree.branches[branch_choice].sections[0].responses.Length;

    switch (size)
    {
        case 1:
            button1.SetActive(true);
            button2.SetActive(false);
            button3.SetActive(false);
            break;
        case 3:
            button1.SetActive(true);
            button2.SetActive(true);
            button3.SetActive(true);
            break;
        default:
            button1.SetActive(true);
            button2.SetActive(true);
            button3.SetActive(true);
            print("Response length of " + interrogation_dialogue_tree.name + "exceeds max limit of 3. setting to default 3 responses");
            break;
    }

    //response_choice = 0;

    /*
    ConfigureInterrogationButton(button1);
    ConfigureInterrogationButton(button2);
    ConfigureInterrogationButton(button3);
    */
    response_choice = 0;
    ConfigureInterrogationButton(button1);
    ConfigureInterrogationButton(button2);
    ConfigureInterrogationButton(button3);
    //button1.GetComponentInChildren<TextMeshProUGUI>().text = interrogation_dialogue_tree.branches[0].sections[0].responses[0].response_dialogue;
    //button2.GetComponentInChildren<TextMeshProUGUI>().text = interrogation_dialogue_tree.branches[0].sections[0].responses[1].response_dialogue;
    //button3.GetComponentInChildren<TextMeshProUGUI>().text = interrogation_dialogue_tree.branches[0].sections[0].responses[2].response_dialogue;

    response_choice = 0;

    vk_button1 = kit.GetComponent<VKKit>().vk_button1;
    vk_button2 = kit.GetComponent<VKKit>().vk_button2;
    vk_button3 = kit.GetComponent<VKKit>().vk_button3;

    ConfigureVKButton(vk_button1);
    ConfigureVKButton(vk_button2);
    ConfigureVKButton(vk_button3);

}
```

As previously mentioned, the on-screen buttons do not help in the progression of the interrogation mini game. Instead, the buttons on the lie-detector machine will progress the mini game. The model of the lie-detector machine was based on the *Voight-Kampff* machine that was used in the *Blade Runner* film. This model will have three buttons, separate from the main model.

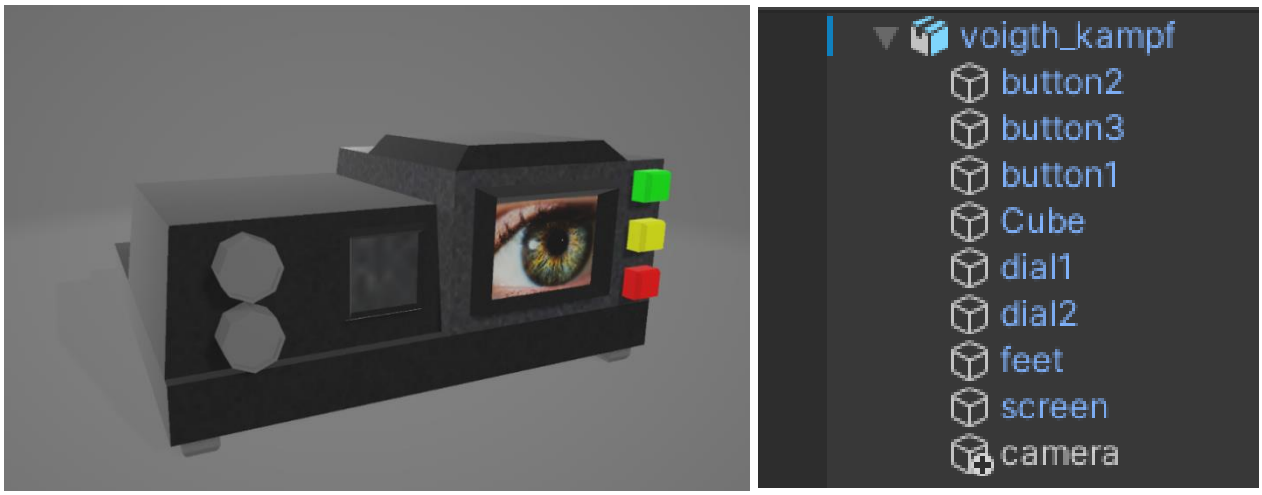


Figure 5.2.2 - Voight Kampff model and model parts.

Like the on-screen buttons, the lie-detector machine buttons are parsed into this method, changing some public variables that will allow progression of the mini game.

```
3 references
void ConfigureVKButton(GameObject vk_button)
{
    vk_button.GetComponent<VKButtons>().interrogation_response_choice = response_choice;
    vk_button.GetComponent<VKButtons>().interrogation_branch_choice = interrogation_dialogue_tree.branches[branch_choice].sections[0].responses[response_choice++].next_branch_id;
}
```

With the buttons now configured, the mini game can be started. When the player starts the mini game, a Boolean called *in_interrogation* is changed from false to true. When this happens, the Dialogue Manager will constantly poll the mouse to check if the let-mouse button was pressed. If it was, it will cast a ray and check if it hit an object and said object contained a script called *VKButton*. The values stored in the script are parsed into the method *UpdateInterrogationConvo*, displaying the new dialogue text and responses.

```
if(Input.GetMouseButtonDown(0))
{
    if (Physics.Raycast(ray, out hit) && hit.collider.GetComponent<VKButtons>())
    {
        int branch_choice = hit.collider.GetComponent<VKButtons>().interrogation_branch_choice;
        int response_choice = hit.collider.GetComponent<VKButtons>().interrogation_response_choice;

        UpdateInterrogationConvo(branch_choice, response_choice);

        print("HIT " + hit.collider.name);
        //Instantiate(debug_cube, new Vector3(hit.point.x, hit.point.y, hit.point.z), Quaternion.Euler(Vector3.zero));

        print(hit.collider.GetComponent<VKButtons>().interrogation_branch_choice);

    }
}
```

The Dialogue Manager now must check if any other conditions were met. If the player passed the *human_type* level but stayed below the *turn_hostile* level when finished, the suspect is then revealed whether if they're real or fake on the screen, using a coroutine. If not, the mini game continues.

```
1 reference
void UpdateInterrogationConvo(int branch_choice, int choice)
{

    if (interrogation_dialogue_tree.branches[this.branch_choice].sections[0].responses[choice].end_on_response)
    {
        print("exited conversation");


        if (talking_npc.GetComponent<AIAgent>().aggression_level >= interrogation_dialogue_tree.reveal_human_type_level && talking_npc.GetComponent<AIAgent>().aggression_level < interrogation_dialogue_tree.turn_hostile_level)
            //  display text on screen to say suspect is human
            if (interrogation_dialogue_tree.is_real_human) StartCoroutine(DisplayConsoleText("suspect_type: [HUMAN]", 3f));
            // display text on screen saying suspect is fake,
            else
            {
                StartCoroutine(DisplayConsoleText("suspect_type: [ERSATZ]", 3f));
                if (interrogation_dialogue_tree.post_interrogation_dialogue != null)
                    talking_npc.GetComponent<AIAgent>().dialogue_tree = interrogation_dialogue_tree.post_interrogation_dialogue;
            }

        else
            StartCoroutine(DisplayConsoleText("suspect_type: [UNKNOWN]", 3f));

        kit.SetActive(false);
        //  turns npc hostile if is aggressive is enabled
        if (talking_npc.GetComponent<AIAgent>().aggression_level >= interrogation_dialogue_tree.turn_hostile_level)
            talking_npc.GetComponent<AIAgent>().is_aggressive = true;



        HideDialogue();
        in_convo = false;
    }
    else
    {

        talking_npc.GetComponent<AIAgent>().aggression_level += interrogation_dialogue_tree.branches[this.branch_choice].sections[0].responses[choice].add_aggression;
        this.branch_choice = branch_choice;
        DisplayInterrogationConvo();
    }


}
```

To display the *human_type* of the suspect is, a small coroutine is created. A coroutine is a method that can pause for a period of time and resume where it left off without causing any performance loss. Creating a Coroutine is very similar to creating a method, but each coroutine must return an IEnumerator somewhere in its body, or else it won't function.

```
4 references
IEnumerator DisplayConsoleText(string text, float time)
{

    console_text.enabled = true;
    console_text.text = text;

    yield return new WaitForSeconds(time);

    console_text.enabled = false;
}
```

A coroutine is started by calling it in the *StartCroutine* function, along with parsing variables when applicable.

```
if (talking_npc.GetComponent<AIAgent>().aggression_level >= interrogation_dialogue_tree.reveal_human_type_level && talking_npc.GetC
    //  display text on screen to say suspect is human
    if (interrogation_dialogue_tree.is_real_human) StartCoroutine(DisplayConsoleText("suspect_type: [HUMAN]", 3f));
    // display text on screen saying suspect is fake,
    else
    {
        StartCoroutine(DisplayConsoleText("suspect_type: [ERSATZ]", 3f));
        if (interrogation_dialogue_tree.post_interrogation_dialogue != null)
            talking_npc.GetComponent<AIAgent>().dialogue_tree = interrogation_dialogue_tree.post_interrogation_dialogue;
    }
```

The result of the mini game will display three variations of this text, either the suspect is a "ersatz", "human" or "unknown". The unknown text is reserved for when the player doesn't exceed the *human_type* level or exceeds the *turn_hostile* level.

Figure 5.2.3 - Suspect type revealed on screen.

With the player now capable of seeing the result of the interrogation mini game, to achieve the goal of a random suspect being the target, a script containing a random number generator must be created. Creating a new interrogation dialogue for a real human, this new dialogue and the existing fake human are stored in a list of type *InterrogationDialogueTreeScriptableObject*. A random number generator seeded with the time since epoch then creates a random number, and an if statement changes the dialogue of the agent if conditions are met.

```
Unity Script (1 asset reference) | 0 references
public class RandomTarget : MonoBehaviour
{
    public AIAgent suspect;

    public List<InterrogationDialogueTreeScriptableObject> interrogation_tree_scripts;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        //Random.InitState(1);
        Random.InitState((int)System.DateTime.Now.Ticks);

        int rand = Random.Range(0, 100);

        if (rand <= 50) suspect.interrogation_dialogue_tree = interrogation_tree_scripts[0];        //  real human
        else suspect.interrogation_dialogue_tree = interrogation_tree_scripts[1];                   //  fake human
    }
}
```

5.3 Interrogation Mini Game UML Diagram

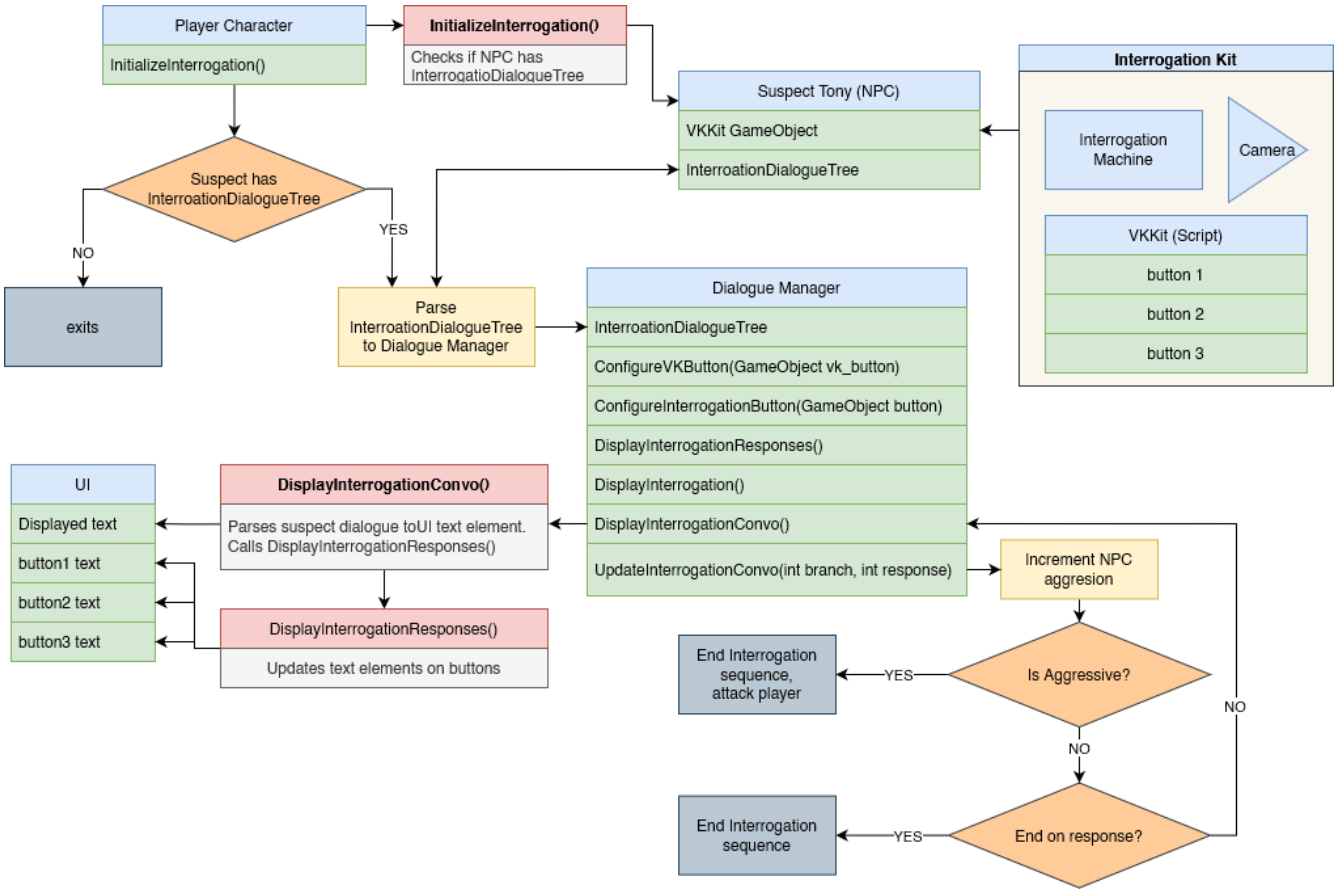


Figure 5.3.1 - UML diagram of Interrogation mini game functionality in Dialogue Manager.

6 Non-Playable Character Finite State Machine

Modern video game artificial intelligence has progressed a lot over the past three decades, from just walking straight towards the player to organically reacting to events from both the player and the game. There are two main ways to creating non-playable character artificial intelligence; state machines and behaviour trees.

State machines, or finite state machines, are the oldest forms of NPC AI in video games, reaching as far back as PAC MAN in 1980 and is still used to this day. Games like DOOM (1993), Half-life (1998), Half-life 2 (2004), F.E.A.R (2006), DOOM 2016, and the modern Tomb Raider series all use Finite State Machines. F.E.A.R famously had very intelligent NPC AI, introducing the first iteration of *Goal Oriented Action Planning* to video games. To simplify how it operates, it states in a 2006 Game Developer Conference paper that (Orkin, 2006):

1. NPC has a state called *hungry*, and its *goal* is to not be hungry.
2. NPC has two options to satisfy state; call for a takeaway or cook meal.
3. NPC can only call for takeaway if they have a phone and can only cook if they have food in the fridge.
4. NPC then checks world state to see if they can execute those actions.
5. If the NPC finds a phone but has no food in the fridge, their plan is to call for a takeaway.
6. If the NPC has food in the fridge but can't find a phone, their plan is to cook food.
7. If both options are available, either goal will satisfy their "plan".

While finite state machines are very useful and simple to make, they come with the drawback that any goal the developers want the NPC to do, they must develop a new state and integrate it into the NPC state machine. This can quickly make it very cluttered and frustrating to debug.

Another alternative that has seen widespread adoption in recent years is the *Behaviour Tree*, popularized by game studio *Bungie* with the release of *Halo 2* in 2004 (Isla, 2006). Like the binary tree, it has a root and several child nodes telling the AI to execute some code when conditions are met. A behaviour tree is easy to implement, more efficient on resources, and easy to expand and reuse, unlike a state machine, hence its much wider industry adoption.

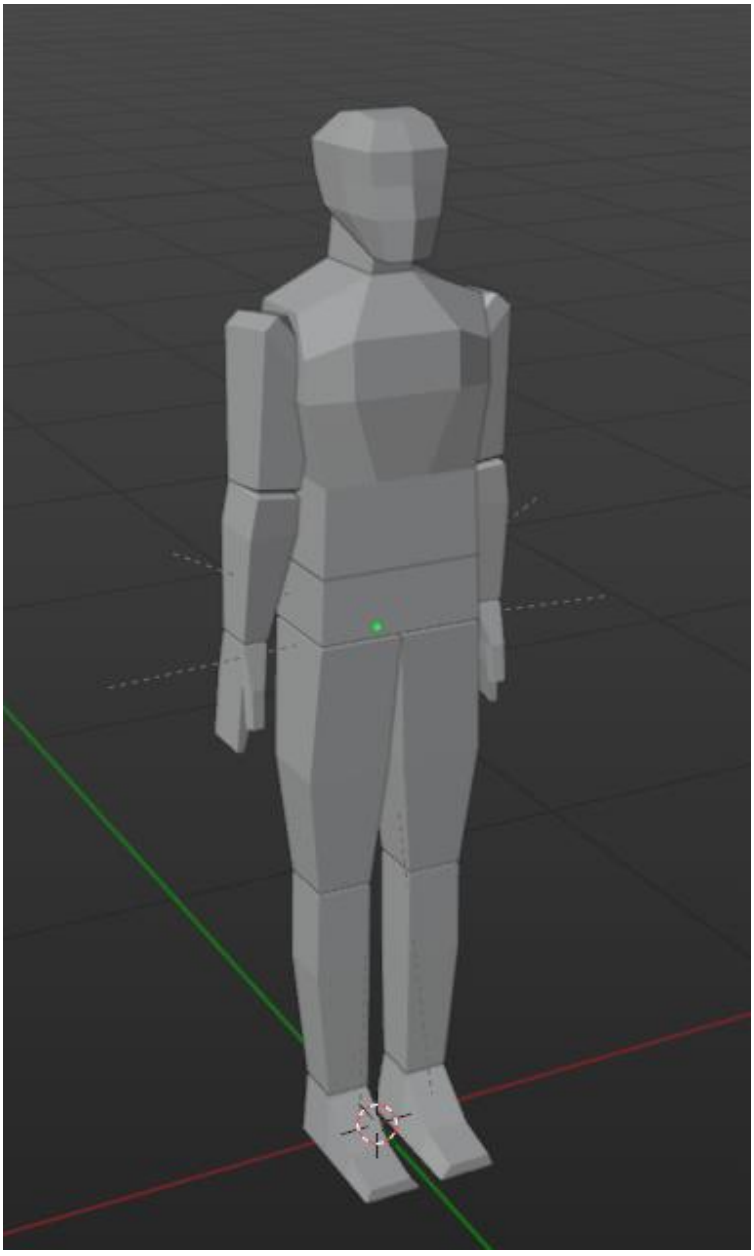
While a behaviour tree makes sense to begin with, I decided to use the state machine as it seemed like an interesting topic to research and implement, and I only needed a handful of states since I had no intention of creating advanced NPC AI to begin with.

To start making a non-playable character, we need to create a model

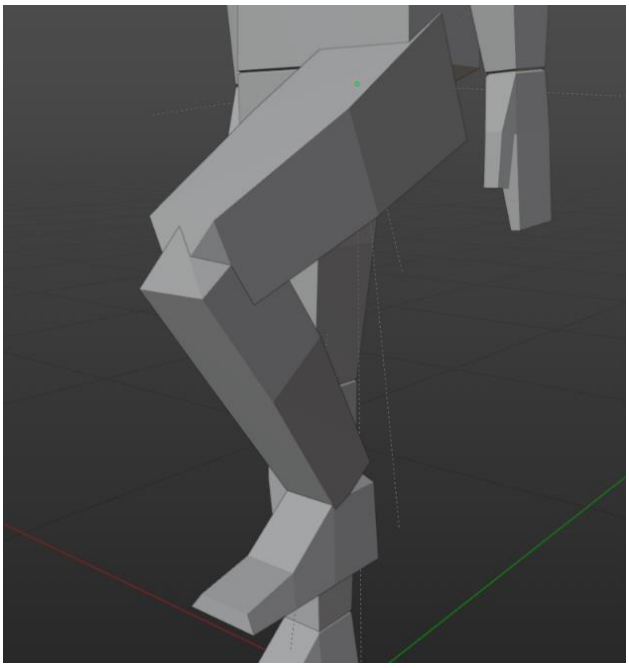
6.1 Non-Playable Character Model

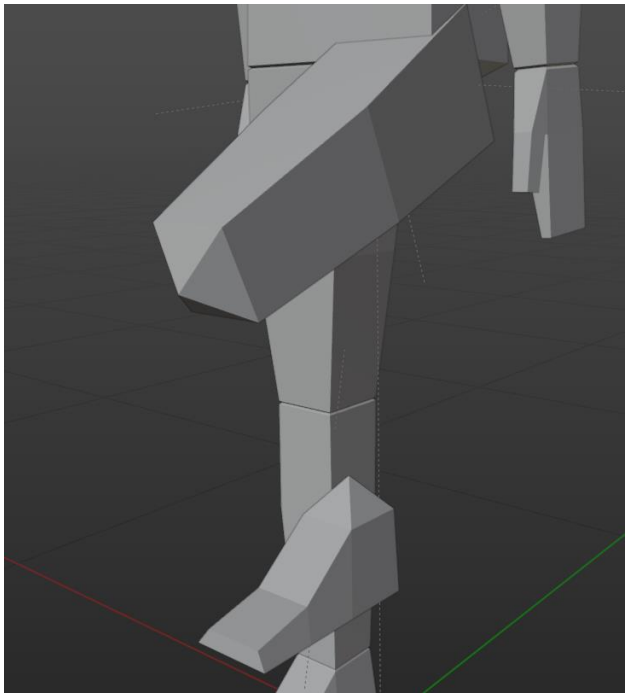
When thinking about the characters the player would meet in a run-down city, my first idea was a group of scavengers, or “scavs” for short. From there, I decided to create a plain default model that I can reshape and change for other characters down the line. A recent design trend in games, and more so in the indie scene, is creating models with a low poly count. Instead of having characters with hyper-detailed designs, a low-poly character is very blocky, often consisting of rectangles.

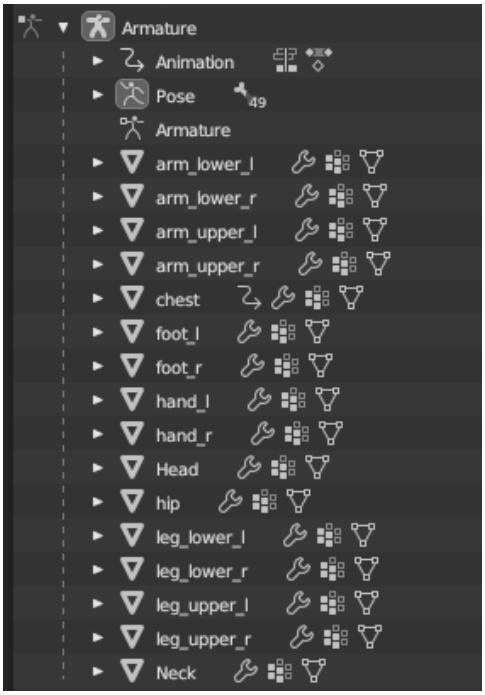
The biggest advantage of having a low-poly character is that it speeds up development time, as you don’t need to create very detailed models.

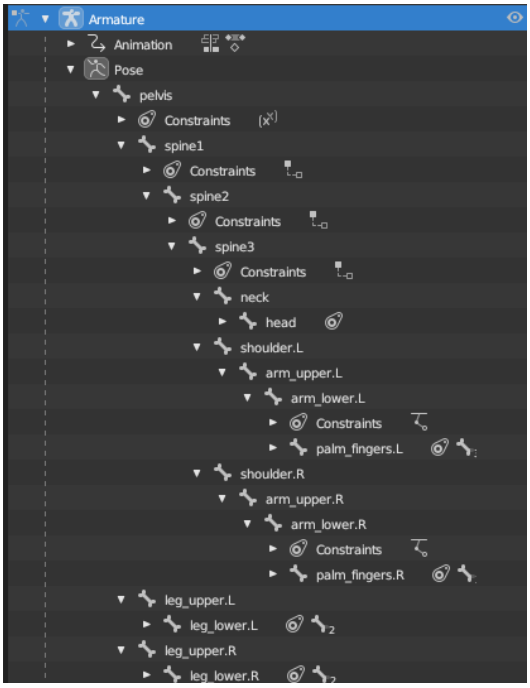


Since the model is composed of multiple meshes, if the upper and lower leg are bent, a very visible flat face would be seen. To mitigate this, we create a vertex or point on the centre of the face and pull it out, creating a pyramid. When the joint is bent, it looks much more appealing.









In order to move and animate the character within the game, we need to add an *armature*, or a set of *bones*. By assigning the meshes to the appropriate bones, moving the upper leg armature will move the upper leg mesh with it. While this does allow movement of the model, there is still more we can do to.

In Blender, a modification to the armature called *Inverse Kinematics* can be applied to a set of specified bones. There are two ways of moving armatures, *Forward Kinematics* and *Inverse Kinematics*. Forward kinematics is a way of moving an armature where parent bones will move child bones, but child bones cannot move parent bones. This is useful in some cases, such as a shoulder, but the most preferable way of moving bones is with Inverse kinematics. Using inverse kinematics, a child bone will influence the angle and position of all parent bones. If an IK constraint is applied to a wrist bone, moving the wrist will automatically calculate the angel of the elbow and shoulder bones. This saves a significant amount of time during the animation process. Instead of meticulously moving each bone in a leg into a walking position using forward kinematics, by applying an inverse kinematics constraint to the ankle, we just need to move the ankle to position, and the knee will automatically move to position.

Finally, to speed up the animations, animator bones that don't deform meshes are created and parented to the existing skeleton. A wrist bone is parented to the hand bone, an ankle bone parented to the foot, and number of other bones such as the head, neck, spine, pelvis, elbows, and knees.

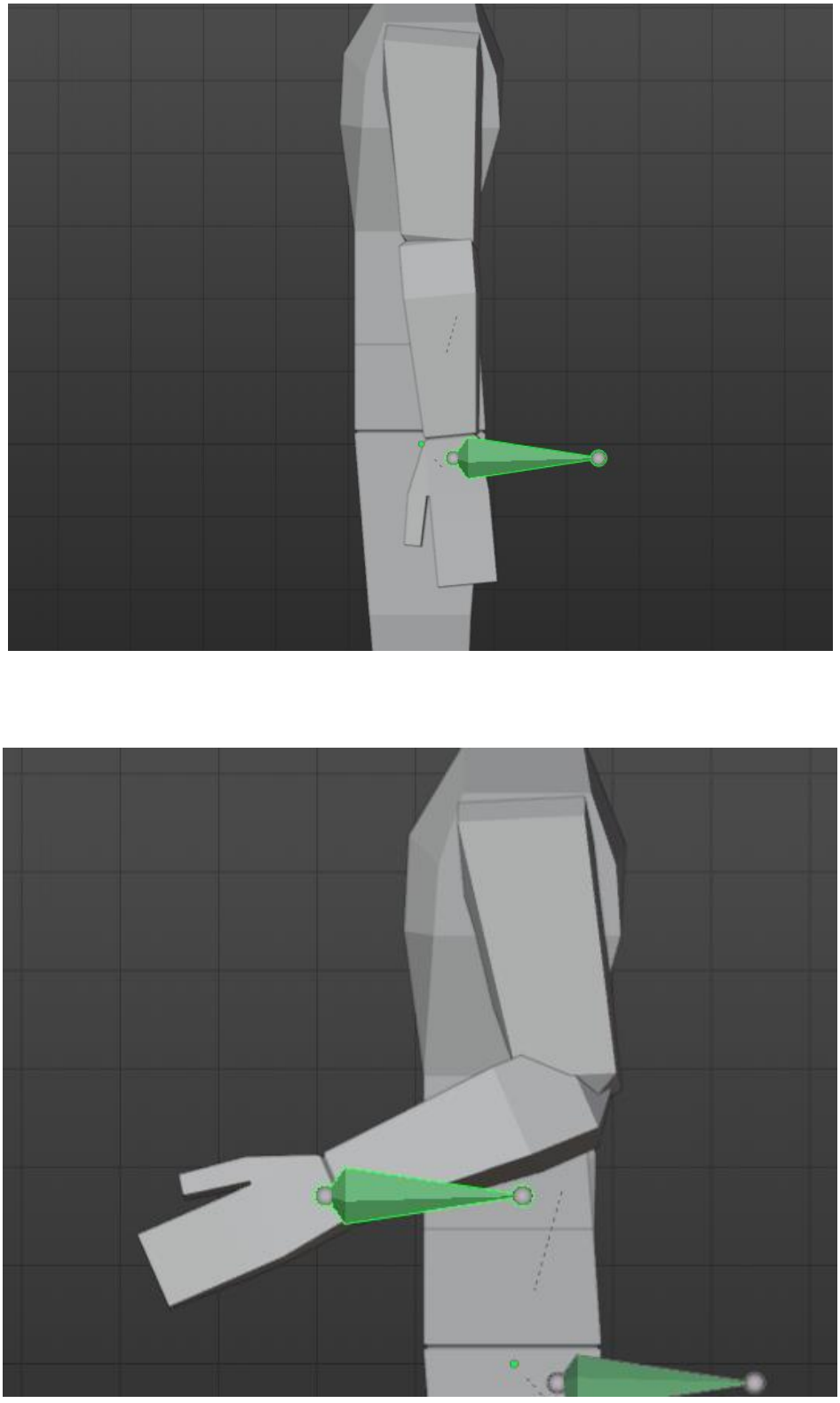


Figure 6.1.1 - Inverse kinematics on wrist bone.

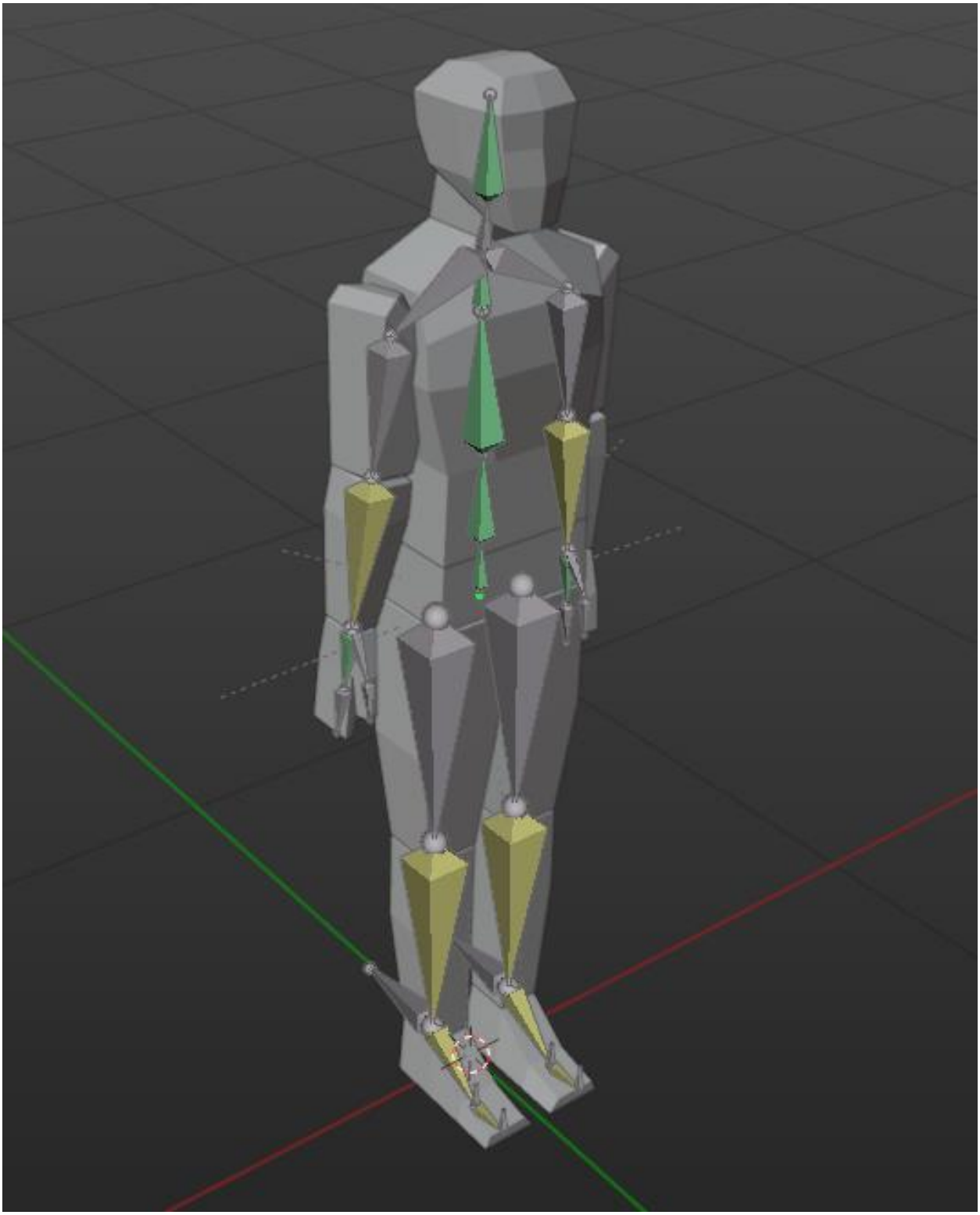


Figure 6.1.2 - Model skeleton with Inverse Kinematics constraints (yellow bones).

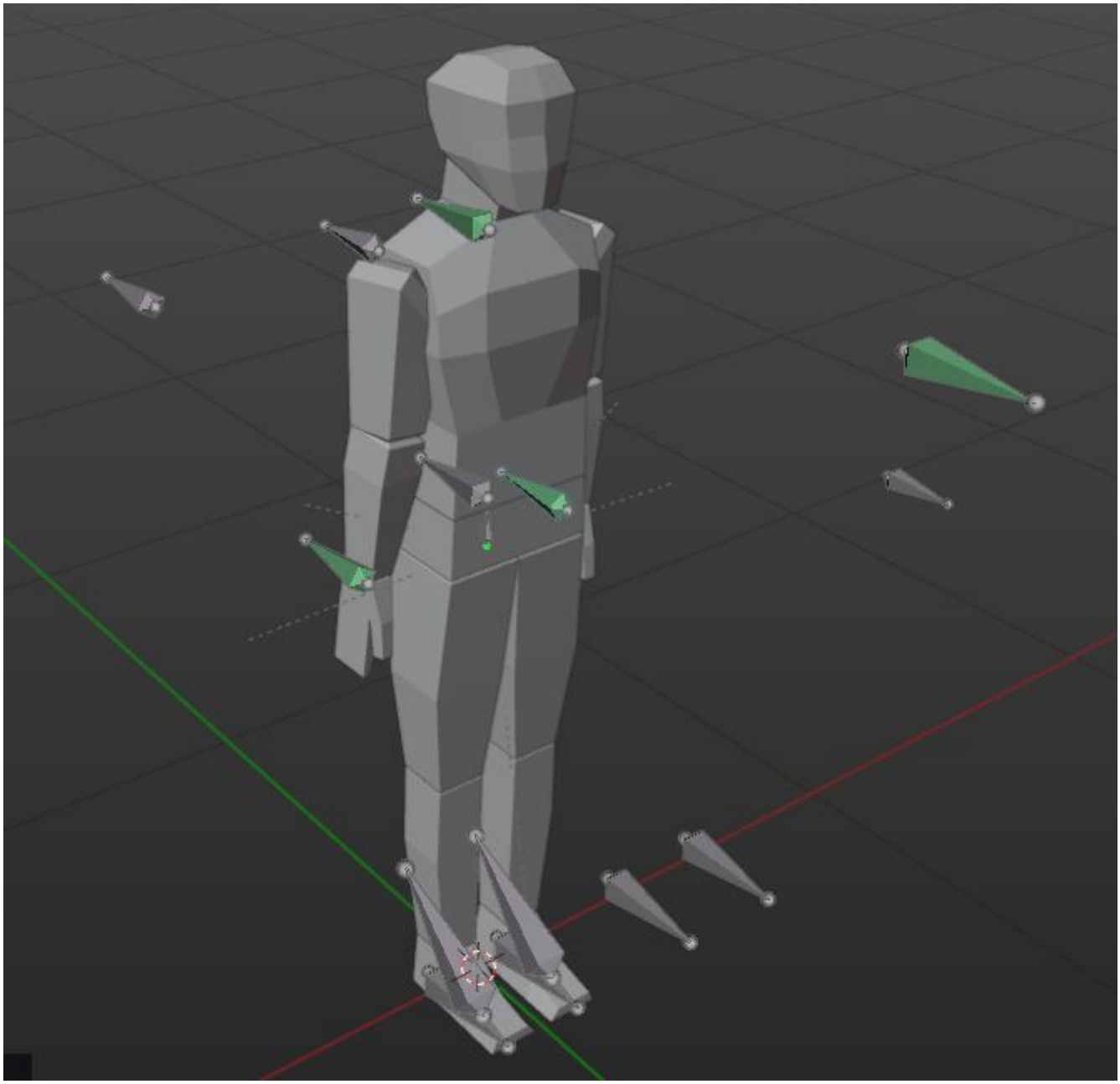


Figure 6.1.3 - Model with animator bones.

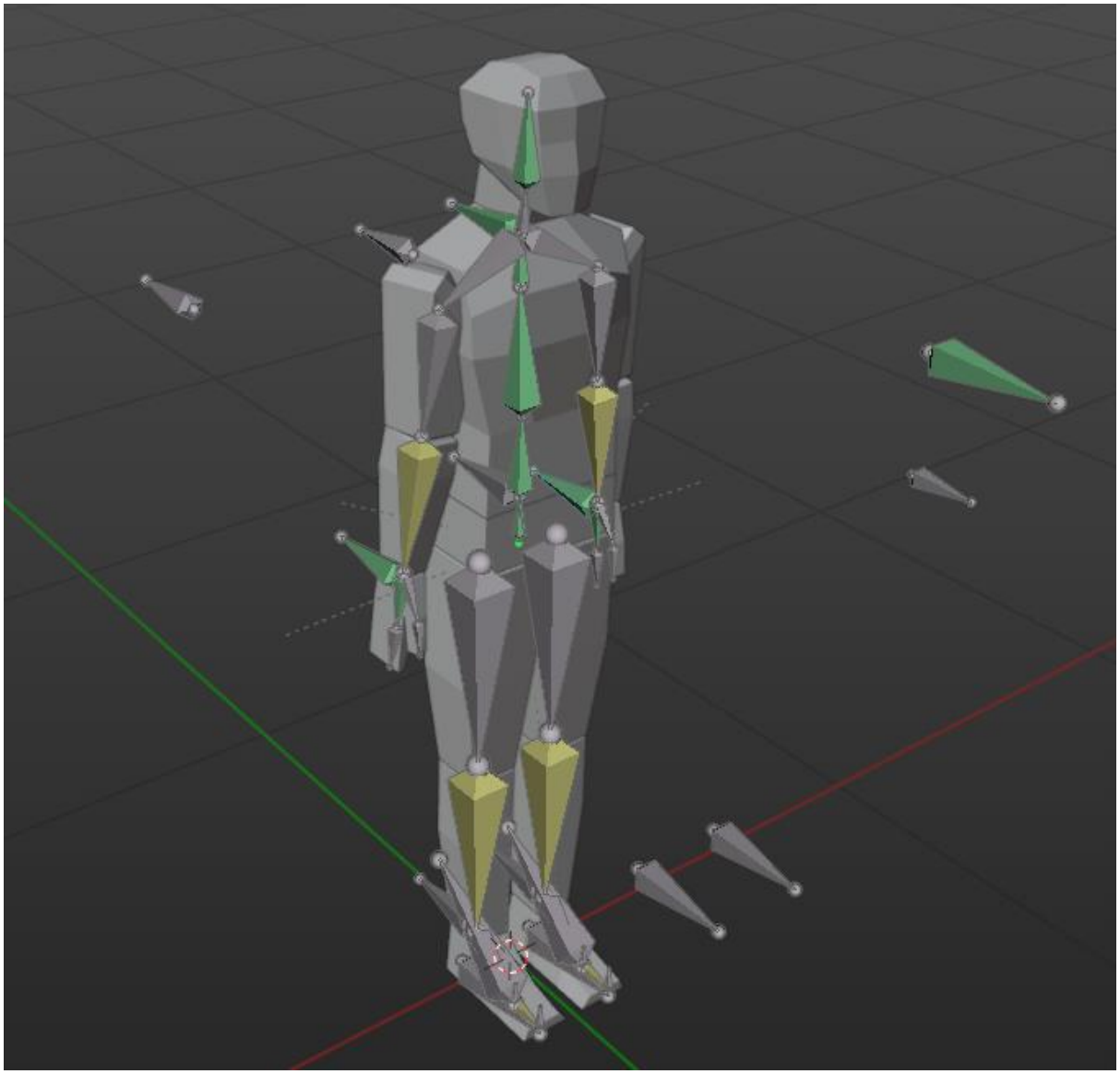
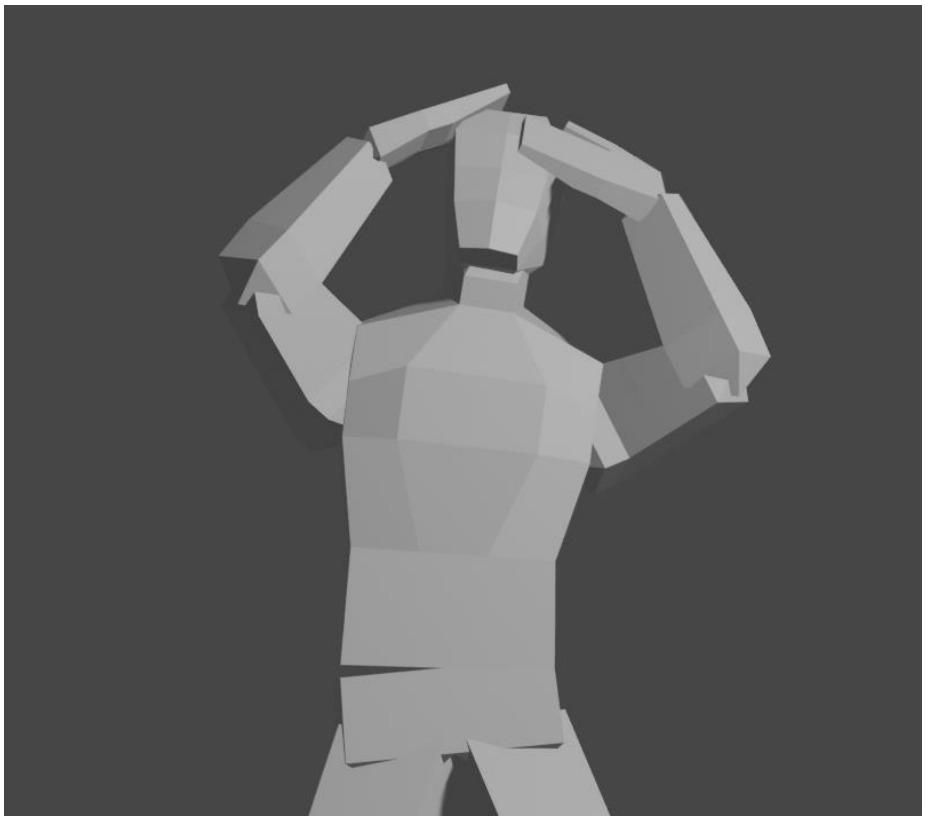
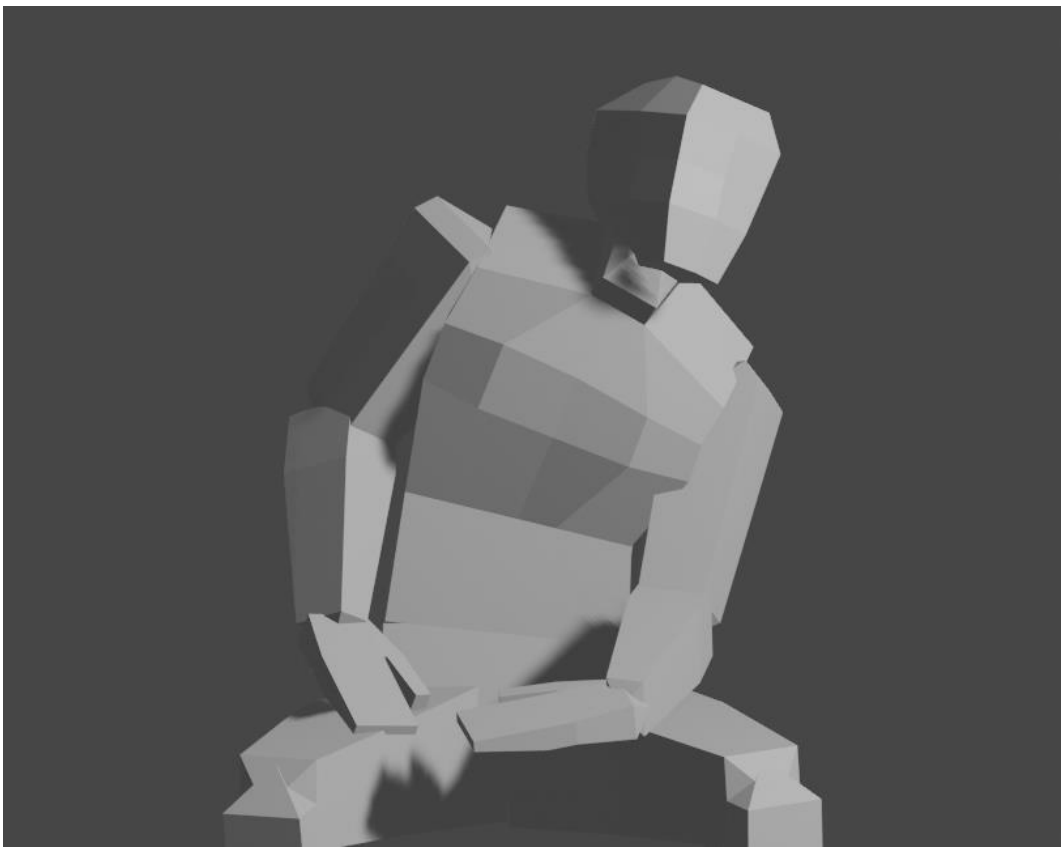


Figure 6.1.4 - Complete skeleton with base and animator bones.

With the animator bones successfully parented, the default low-poly model can now be animated into short animations and poses.





6.2 Non-Playable Character Interactivity

Exporting the character model from Blender to Unity is very easy. Unity supports two file formats natively: OBJ and FBX. OBJ is used for small simple objects such as mugs, phones, or crates. OBJ doesn't support individual meshes, textures, animations, and armatures, thus making the exported file much smaller.

FBX, on the other hand, supports individual meshes and armatures, making it a necessity for models like character models or models with animations. With the model successfully imported into Unity, we can begin giving the model a ragdoll, health, and movement capabilities.

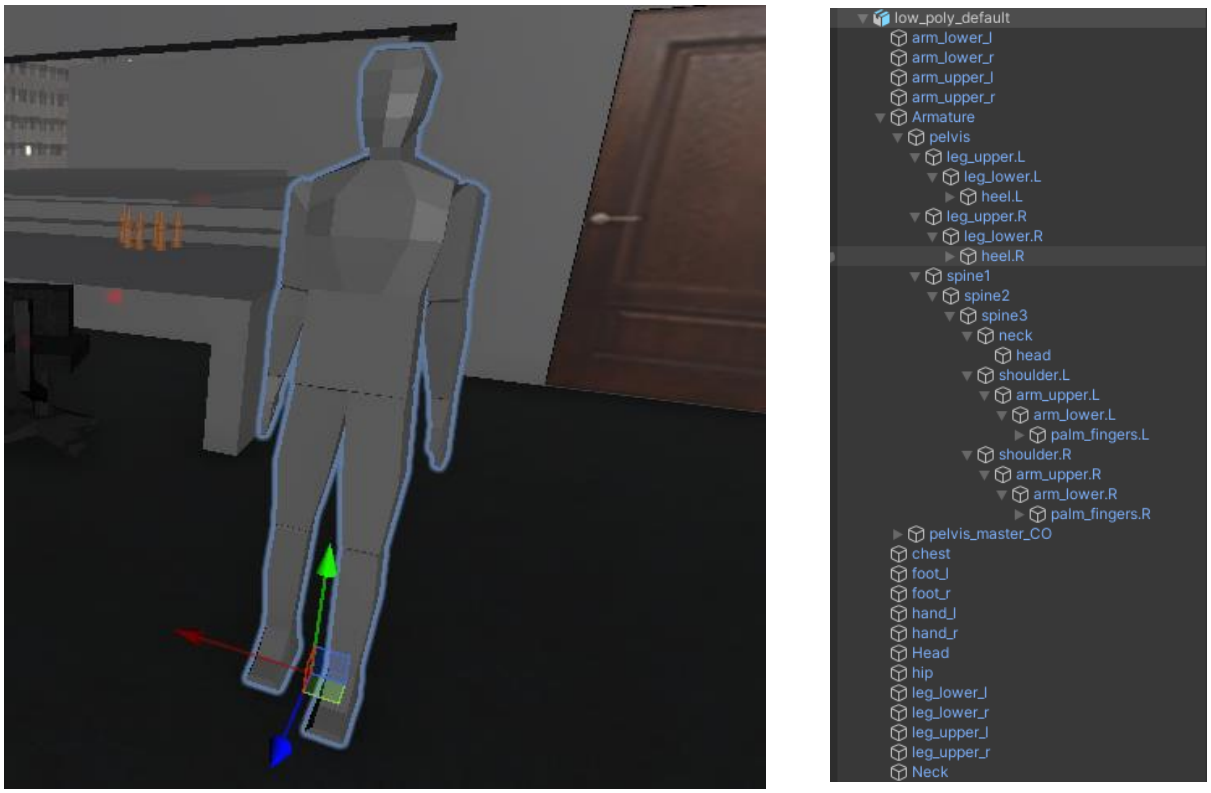
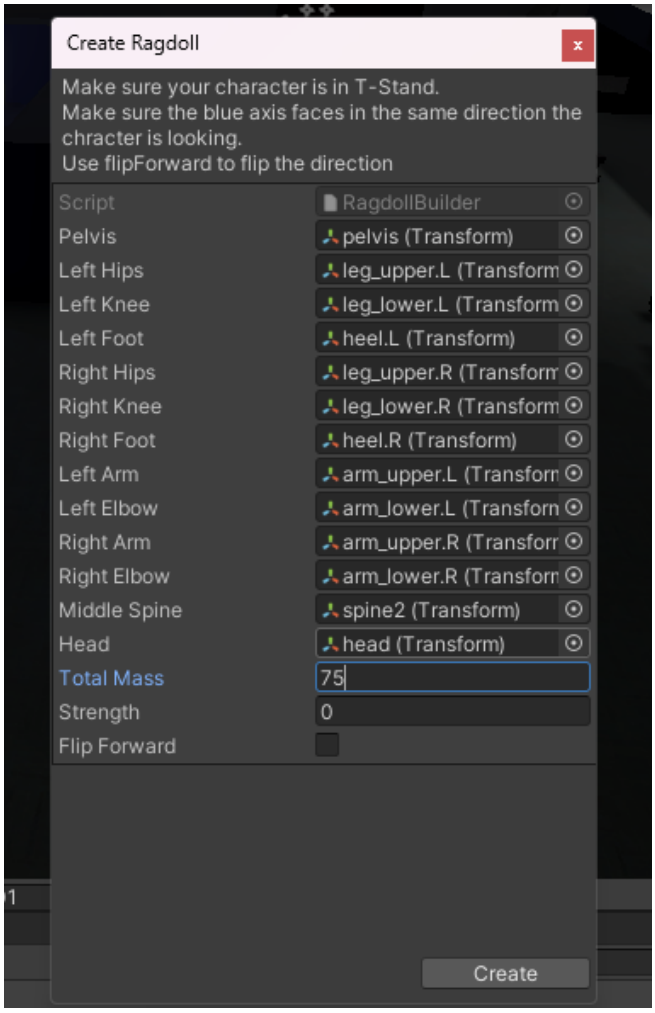
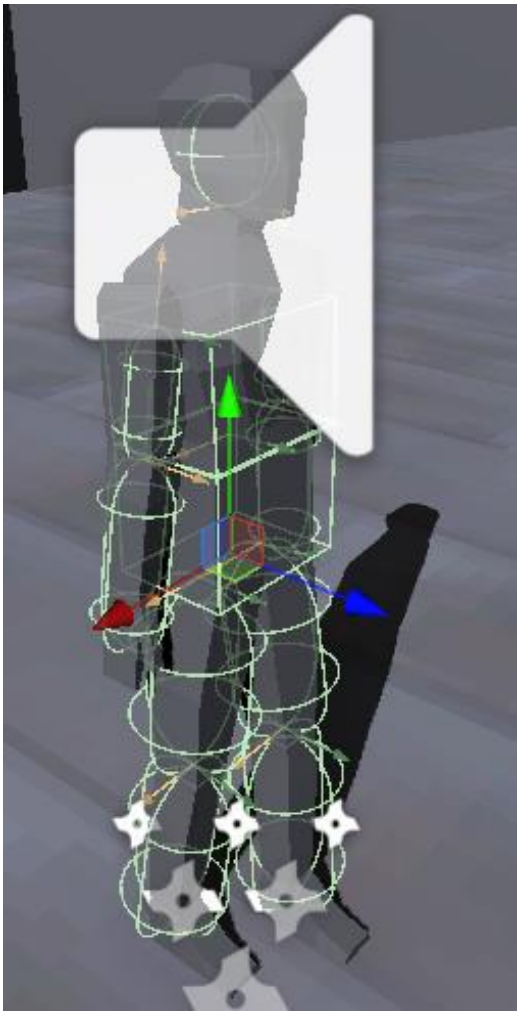


Figure 6.2.1 - Character model in Unity Editor.

In video games, there are multiple ways of showing a character's death, either playing an animation of them falling, turning into a ragdoll, or a mix of both. Playing an animation is very easy for this but out of personal preference, I decided to use a ragdoll. A ragdoll is when all the bones in a model has physics applied to them. This creates a comedic effect of a model losing all strength in their limbs and falling flat on their face or back. This would also be a nice addition for future additions like stealth, where an enemy will be alerted if they see a body, so the player will have to drag the body to a place the enemies can't see.

Unity has a built-in wizard to help automatically build a ragdoll from a character model. When the fields are correctly assigned to the parts of the model armature, they should look like this:





Although this is very simple, issues did arise during development where the armature scale was set to “100” when the correct value is just “1”. It took a bit of time to figure out that the rotation of the armature and the scale of the models can cause issues with the exported model, causing this issue. On the surface, it seems harmless, but this was likely the reason why weapon models were massive when parented to the model.

With the ragdoll completed, the health script can be created. This health script will have a method that will check all the bones in the armature to see if any have a rigidbody. If they do have a rigidbody component but don't have a script called *EntityHitbox*, it will add a *EntitiyHitbox* script to the bone along with a reference to health script. This will let use create localized damage areas, such as the head bone which has a damage multiplier of 5 while the rest of the bones have a damage multiplier of 1. The only exception to this is the scav leader who's head bone has a damage multiplier of 0.75, just to surprise the player.

```
1 reference
void CreateEntityRagdoll()
{
    rigidbodies = GetComponentsInChildren<Rigidbody>();
    foreach (Rigidbody rb in rigidbodies)
    {
        EntityHitbox entity_hitbox;


        //  split into if statemment to prevent adding second hitbox to enemy rigidbody
        if (!rb.GetComponent<EntityHitbox>())
        {
            entity_hitbox = rb.gameObject.AddComponent<EntityHitbox>();
            entity_hitbox.health = this;
        }
        else
        {
            rb.GetComponent<EntityHitbox>().health = this;
        }
    }

}
```

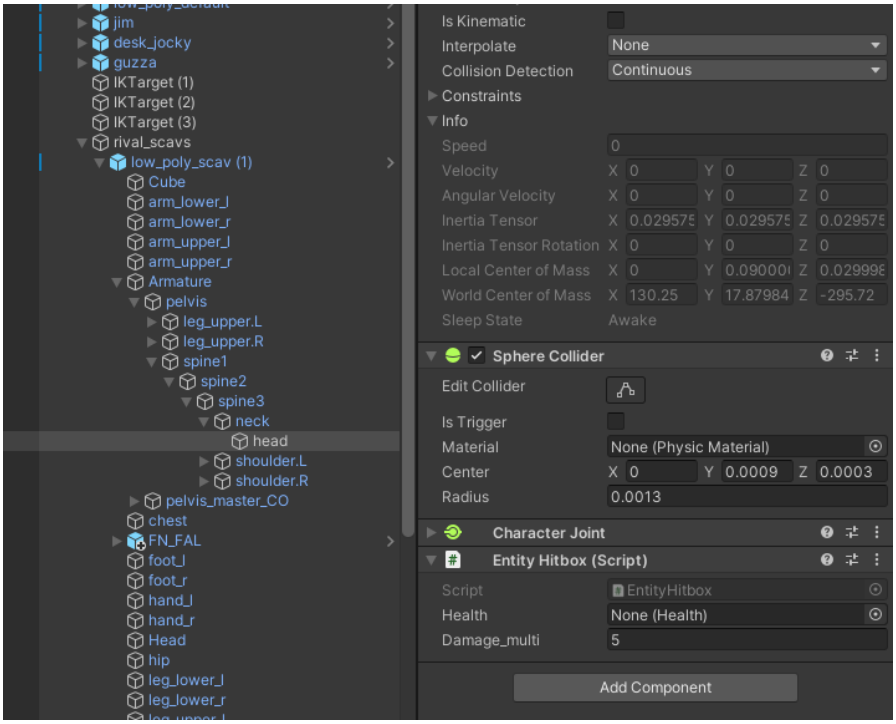


Figure 6.2.2 - Head hitbox for scav NPC has damage multiplier of 5.

With the NPC ragdoll finished, we can move onto NPC AI and locomotion.

6.3 NPC Artificial Intelligence and Locomotion

As mentioned at the beginning of this section, the type of artificial intelligence we'll use is a finite state machine. To begin this, we need to make a state machine class for the NPCs to use.

```
3 references
public class AIStateMachine
{
    public AIState[] states;
    public AIAgent agent;
    public AIStateID currentState;

    1 reference
    public AIStateMachine(AIAgent agent)
    {
        this.agent = agent;

        int numStates = System.Enum.GetNames(typeof(AIStateID)).Length;
        states = new AIState[numStates];
    }

    5 references
    public void RegisterState(AIState state)
    {
        int index = (int)state.getID();
        states[index] = state;
    }

    3 references
    public AIState getState(AIStateID stateID)
    {
        int index = (int)stateID;
        return states[index];
    }

    1 reference
    public void Update()
    {
        getState(currentState)?.Update(agent);
    }

    14 references
    public void ChangeState(AIStateID newState)
    {
        getState(currentState)?.Exit(agent);
        currentState = newState;
        getState(currentState)?.Enter(agent);
    }
}
```

For the state machine class, we need to register a new state using *RegisterState*, change to another state using *ChangeState*, and to update the state, a new AIState method called *getState* is created so that in *Update,* we get the state of on agent and update it. This forms the basis of the state machine the NPCs will use.

To get the NPC agents to use the state machine, a new interface called AIState along with an enum to contain the state IDs are created

```
31 references
public enum AIStateID
{
    Idle,
    Patrol,
    ChasePlayer,
    FindWeapon,
    AttackPlayer,
    Death
}

10 references
public interface AIState
{
    7 references
    AIStateID getID();
    7 references
    void Enter(AIAgent agent);
    7 references
    void Update(AIAgent agent);
    7 references
    void Exit(AIAgent agent);

}
```

Figure 6.3.1 - AIState interface and AIStateID enum.

When creating a new state, we create a new C# script using a template name of "AI[INSERT STATE ACTION]State" like *AIChasePlayerState*. This helps distinguish it from other scripts. To implement the interface, the "monobehaviour" base class at the top of the script must be replaces with *AIState*.

To create a small state that chases the player, a new component called a *NavMeshAgent* must be applied to the NPC. A NavMesh, or Navigation Mesh, lets agents move around the level if the point they wish to travel to is located on the NavMesh. This means static objects like cars or statues put a hole in the nav mesh, preventing the NPC from moving through them as though they never existed.

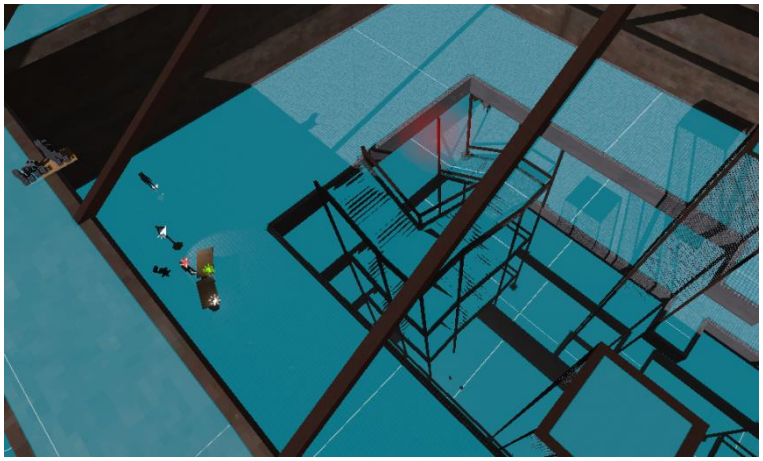


Figure 6.3.2- Blue area means NPC can travel in this area.

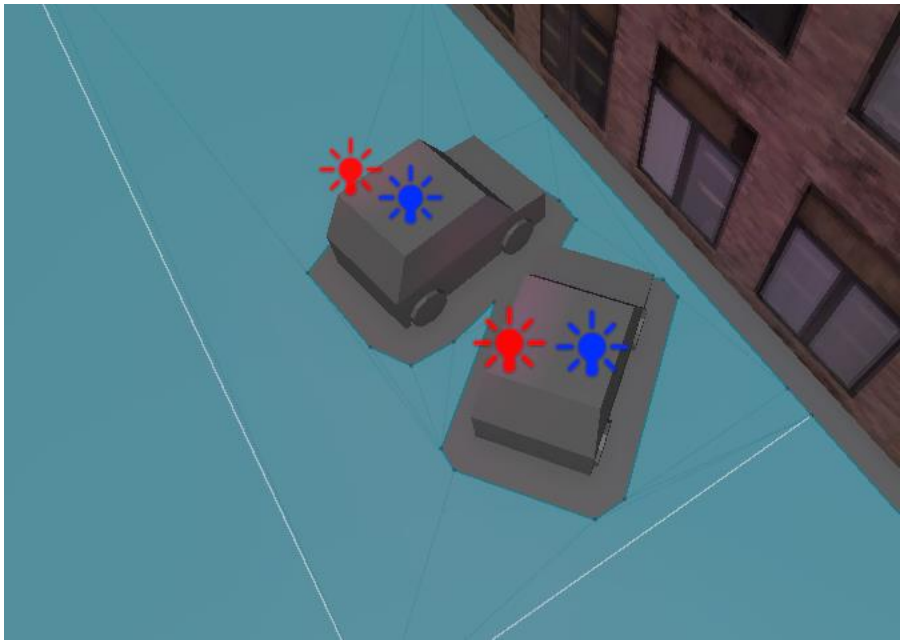


Figure 6.3.3 - Police cars putting holes in NavMesh, preventing NPC moving into those areas.

Adding the NavMeshAgent component to the NPC, we must also reference it within the AIAgent script. By referencing it within the AIAgent script, the AIChasePlayerState can access the component and plot out a path to the player every time the Update function is called. To improve the performance of the game, instead of creating a new path every Update, we create a new path every fifth of a second. Creating a new path is quite intensive, especially if multiple agents are calling it.

```
1 reference
public class AIChasePlayerState : AIState
{

    float timer;

    2 references
    public AIStateID getID()
    {
        return AIStateID.ChasePlayer;
    }

    2 references
    public void Enter(AIAgent agent)
    {
        //agent.branch_dialogue.enabled = false;
    }

    2 references
    public void Exit(AIAgent agent)
    {

    }

    2 references
    public void Update(AIAgent agent)
    {
        timer -= Time.deltaTime;
        if (timer < 0f)
        {
            if (agent.navMeshAgent.enabled)
            {
                float sqdistance = (agent.player_transform.position - agent.navMeshAgent.destination).magnitude;

                if(agent.player_gameobject.GetComponent<playerHealth>().health > 0f)
                {
                    if (sqdistance > agent.config.maxDistance * agent.config.maxDistance)
                    {

                        agent.navMeshAgent.destination = agent.player_transform.position;
                    }

                }
                else
                    agent.stateMachine.ChangeState(AIStateID.Idle);


                timer = agent.config.maxTime;
            }

        }
```

Figure 6.3.4 - AIChasePlayerState script.

If the NPC were to ragdoll while the NavMeshAgent component is still active, an amusing but unwanted bug can occur where the ragdoll will follow the player continuously. To prevent this, we get a reference to the NavMeshAgent component in a new script called *AIDeathState* and disable the component. To change the state, a new condition must be met where if the NPC's health is below a threshold such as "0", the state machine will change the state of the agent to the death state.

```
1 reference
public class AIDeathState : AIState
{

    float time = 0;
    2 references
    public void Enter(AIAgent agent)
    {
        Debug.Log(agent.name + " entered state: AIDeathState");

        agent.npc_audio_source.Stop();

        agent.ragdoll.ActivateRagdoll();
        agent.impact_direction.y = 1f;
        agent.ragdoll.impact_body_part = agent.hit_rb;
        //agent.ragdoll.ApplyForce(agent.impact_direction * agent.config.impact_force, agent.health.death_force_mode);


        agent.ai_weapon.DropWeapon();

        agent.ai_weapon.enabled = false;

        //  ai_weapon_ik must be disabled first as errors prevent rest being disabled
        //  don't fucking touch please
        agent.ai_weapon_ik.enabled = false;


        agent.mesh_sockets.enabled = false;
        agent.enemy_los.enabled = false;
        agent.enabled = false;
    }

    2 references
    public void Exit(AIAgent agent)
    {

    }

    2 references
    public AIStateID getID()
    {
        return AIStateID.Death;
    }

    2 references
    public void Update(AIAgent agent)
    {

    }
}
```

Now then the NPC's health has dropped below the threshold, the NavMeshAgent component will be disabled, and the NPC will enter the death state, activating their ragdoll.

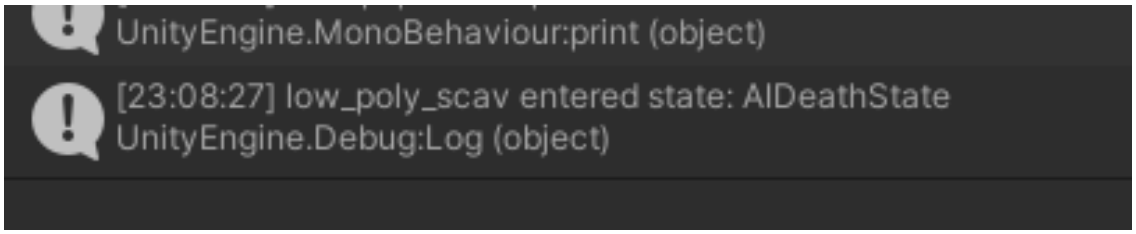


Figure 6.3.5 - Unity Editor console displaying current state of NPC low_poly_scav.

Figure 6.3.6 - NPC after activating its ragdoll.

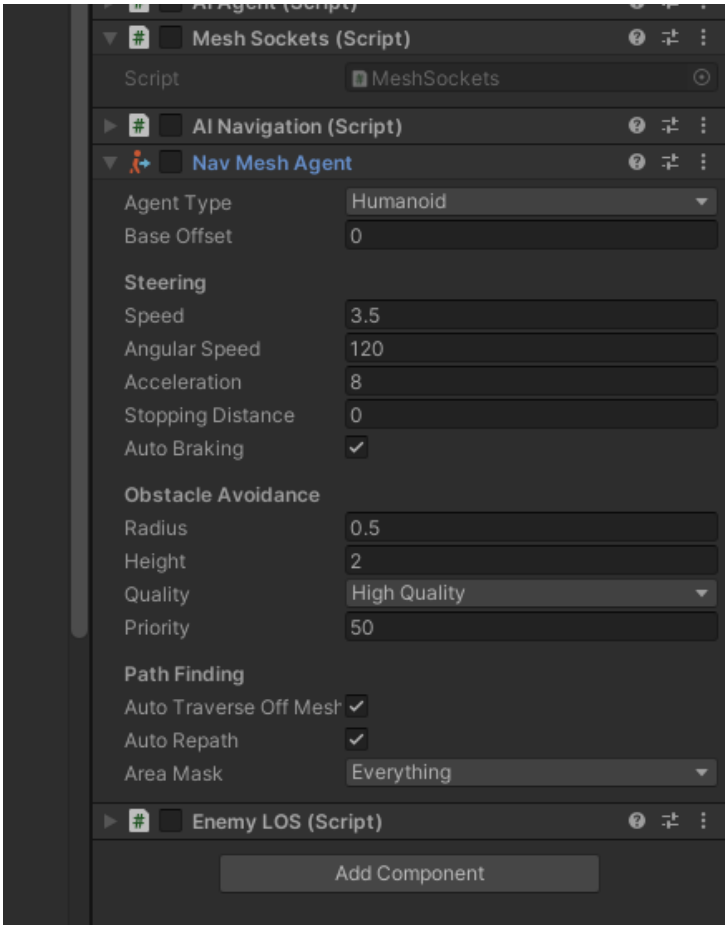


Figure 6.3.7 - Nav Mesh Agent component disabled.

Finally, to add challenge to the game, the NPCs will also attempt to attack the player when provoked or turn hostile when in line of sight. When the NPC is attacked by the player, a Boolean called *is_aggressive* is changed from false to true. This causes the NPC to change to the attack player state. When they entered the attack player state, they check if: is player in range or NPC has weapon. The state machine checks:

1. If attacked, check if weapon is on body.
2. If weapon is attached to body, equip it. Else, plot path to nearest weapon.
3. If weapon is in hand, check if in range. If not in range, chase player, else, attack player.
4. If player health is above threshold, continue attacking, else, enter idle state.

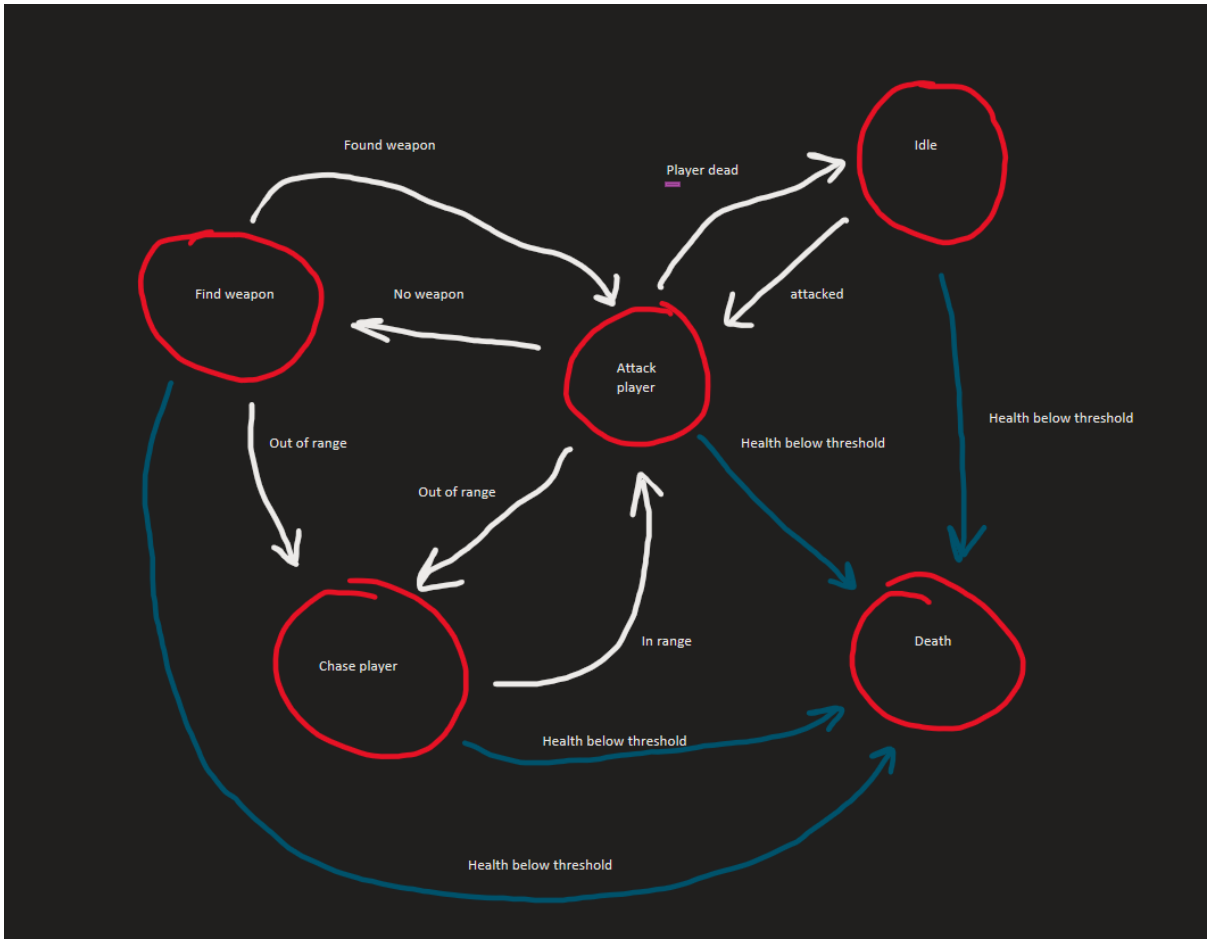


Figure 6.3.8 - Finite State Machine diagram of NPC states.

7 Introduction Level

The introduction level stayed mostly the same from beginning to the final iteration, consisting of desolate buildings, a police station to get information, a dilapidated building containing the scavs, and a large skyscraper where the suspect is kept. The initial version was split into two scenes, a hub world where the player could explore and talk to the police chief, and another level in a skyscraper where the player would need to fight through a few scavs and progress through the crumbling building.

The idea of having a hub world where the player could explore and talk to other NPCs to gather information was an enticing idea but as the project progressed, it didn't seem like a good idea, especially with the theme of player choice and lack of level design experience.

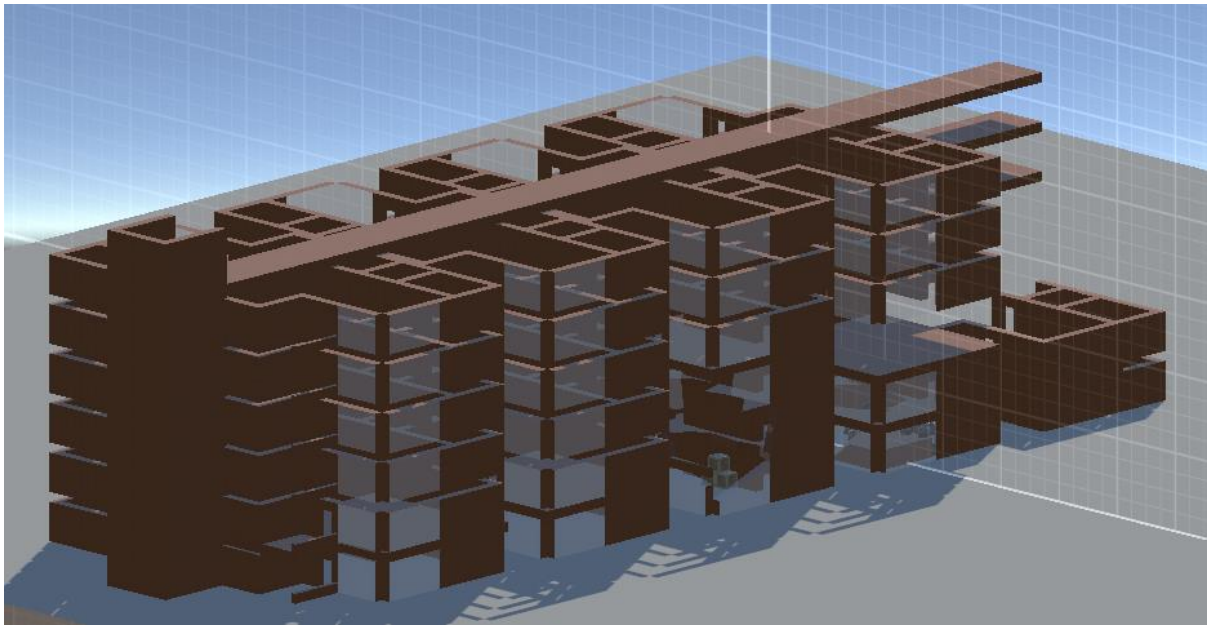


Figure 7.1- Skyscraper where suspect was located.

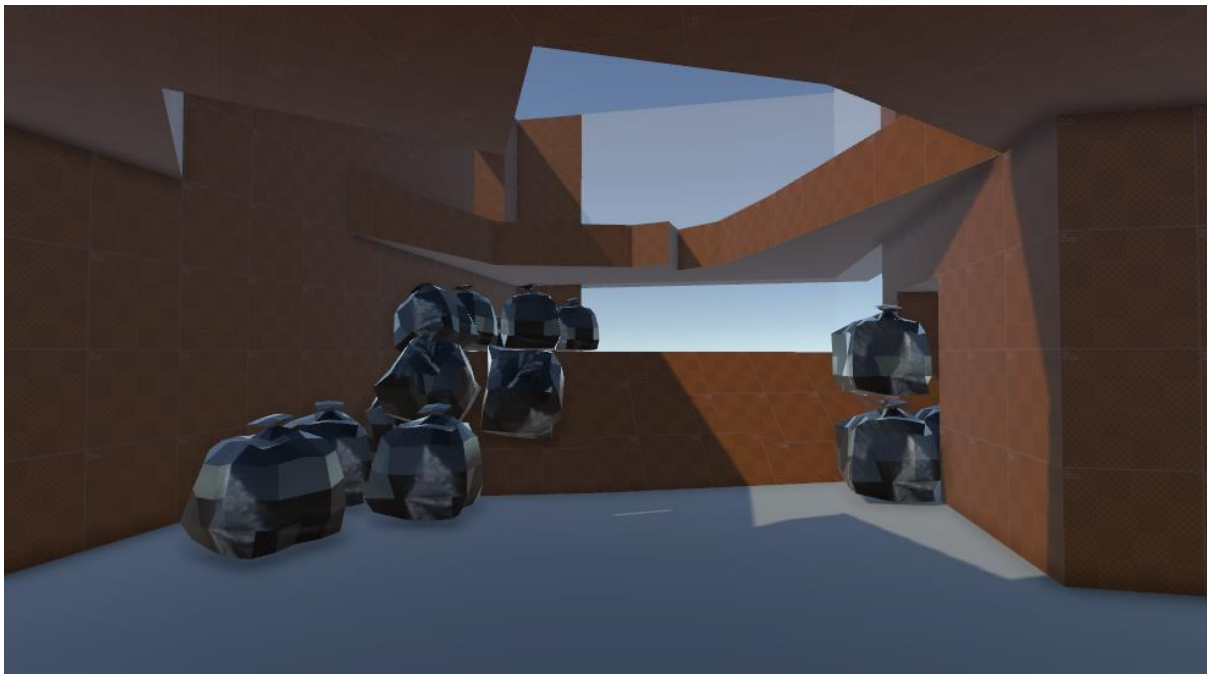


Figure 7.2 - Player could climb up to the first floor with the bon bags or through a door hidden behind the bin bags.

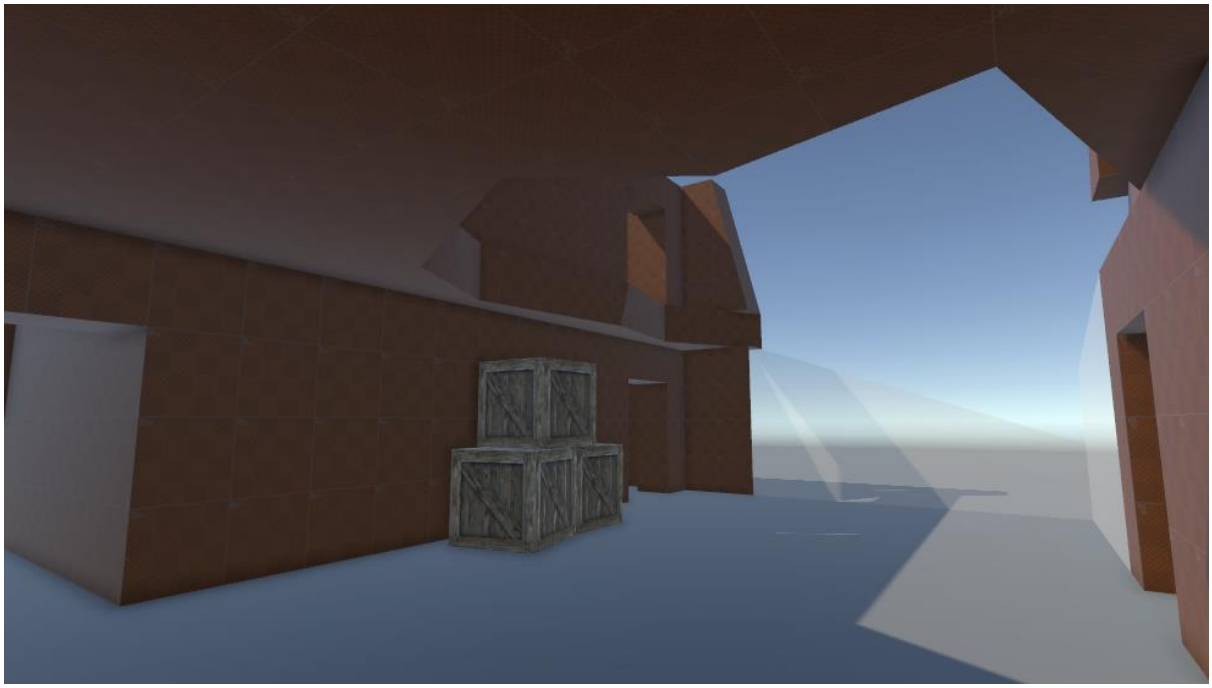


Figure 7.3 - Another route to the first floor by stacking boxes to build stairs.

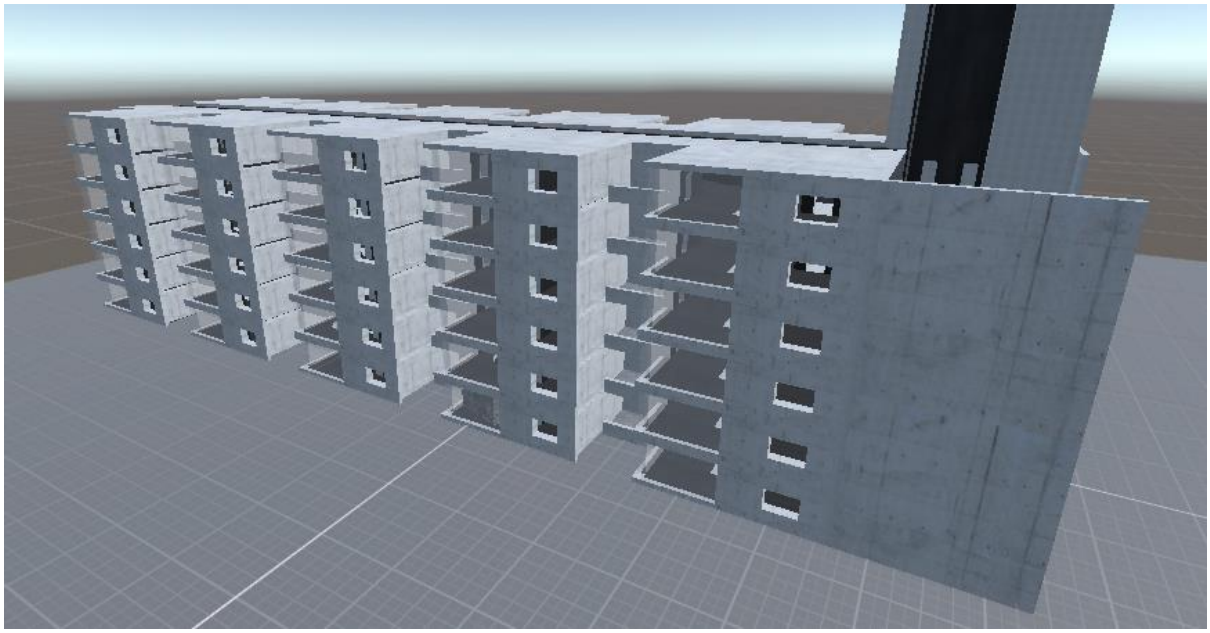


Figure 7.4 - Final iteration of skyscraper.

When designing the level, I was quite happy with the design, but when playtesting it, I instead had mixed feelings about it. While verticality was good, it felt cramped and uncomfortable since the only places the player would travel in were two rooms and a corridor. It was too repetitive for what I wanted. Unfortunately, there was also an issue with Git where the references broke and this level was unsalvageable, despite fixing the issue.

After losing the level, I started designing a new one but instead of having two separate levels, I would combine the hub world level with the skyscraper level. This meant I could spend my time on one level then trying and failing to evenly distribute it on two separate levels.

With the new experience I gained from designing the old levels, I decided that the new one should be smaller and denser to keep with the design goal of player choice.

7.1 Designing New Introduction Level

When starting the game, I wanted the player to start in the bedroom of their apartment. This would let me show the player the basics of the game, how to move, and how to interact with objects around the level. To help with this, I also added box triggers that when the player enters them, some text would display on the top right corner to help guide the player more.

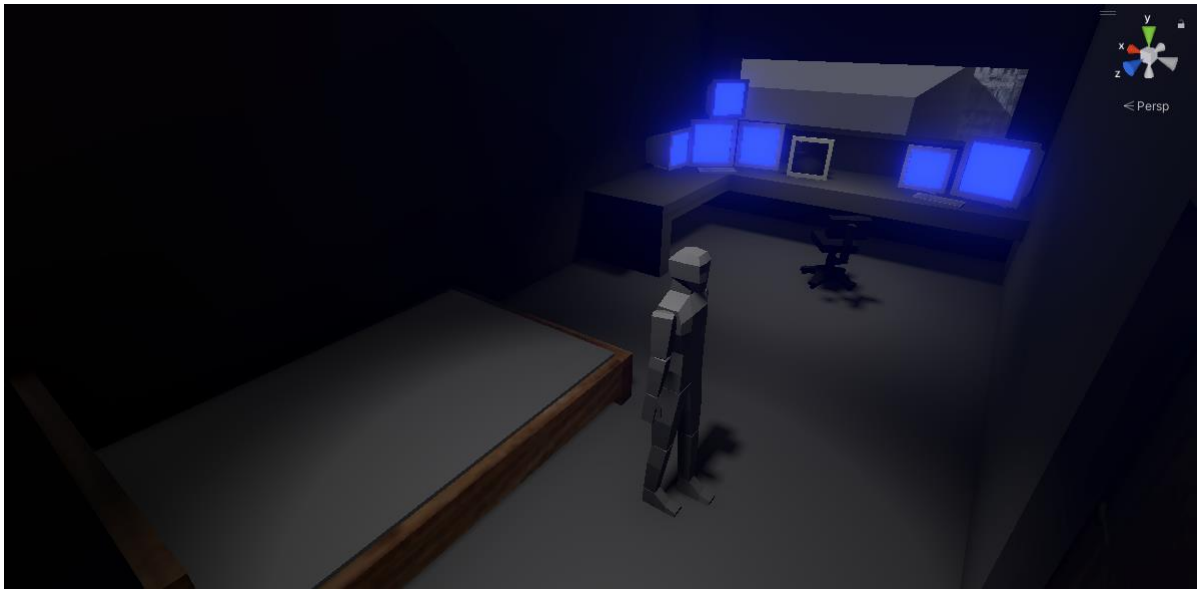


Figure 7.1.1- Player character and bedroom where player starts.

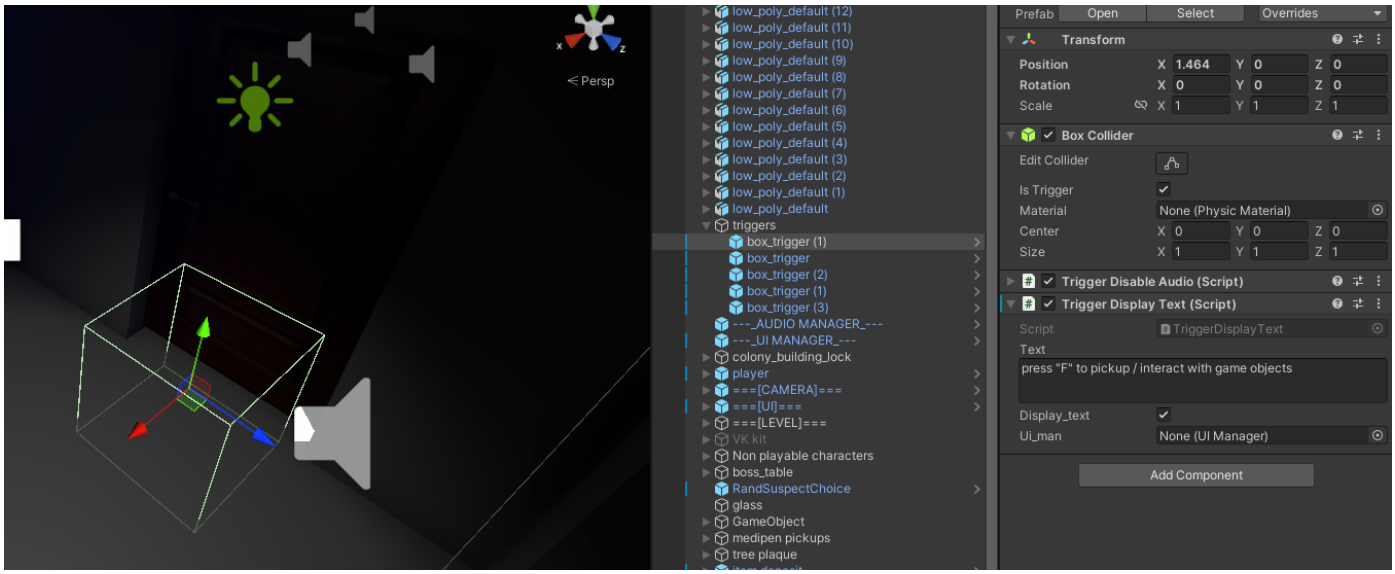


Figure 7.1.2 - Box collider set as trigger so when player enters, text in component is displayed on screen.

To highlight to the player that there are also secret areas to find new weapons and items, I added a small compartment to the storeroom hidden with a sliding door. Inside this compartment is a selection of weapons for the player to pick up and use for later.

Figure 7.1.3 - Orange highlighted door is entrance to storeroom.

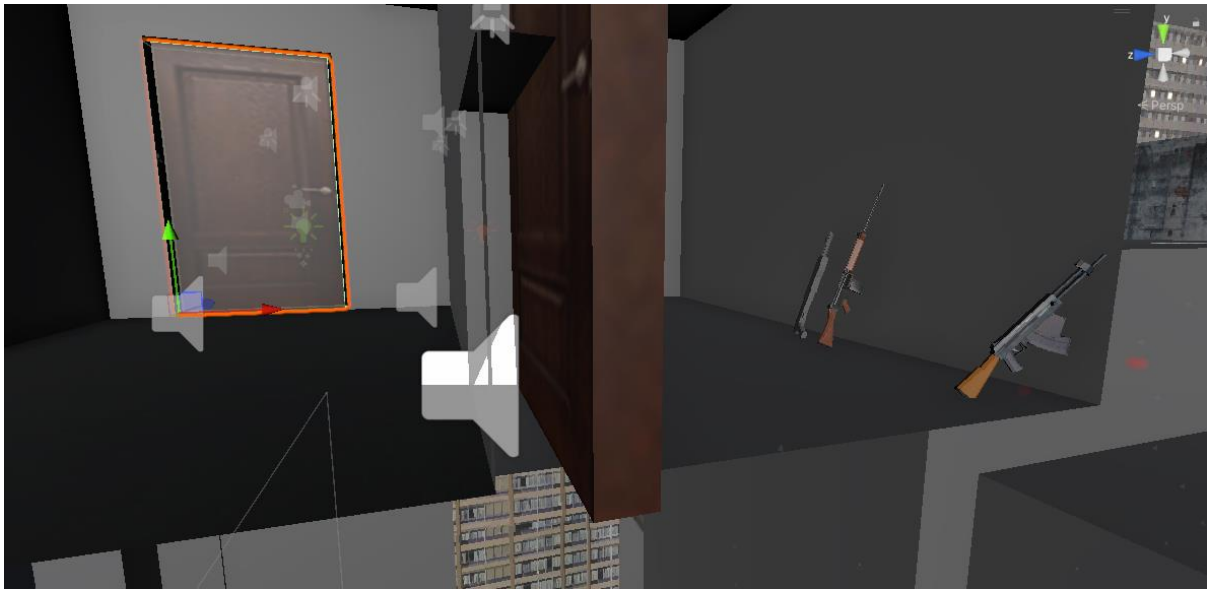


Figure 7.1.4 - Sliding door that reveals weapons for player to pick up.

A box on the corner is used to help the player learn how the dialogue system works, by pressing “F” on it, a conversation will be initialized. This is a slightly altered version of how the dialogue with an NPC works where the Dialogue Manager checks if the entity, we’re talking to has a component called *TalkableEntity*.

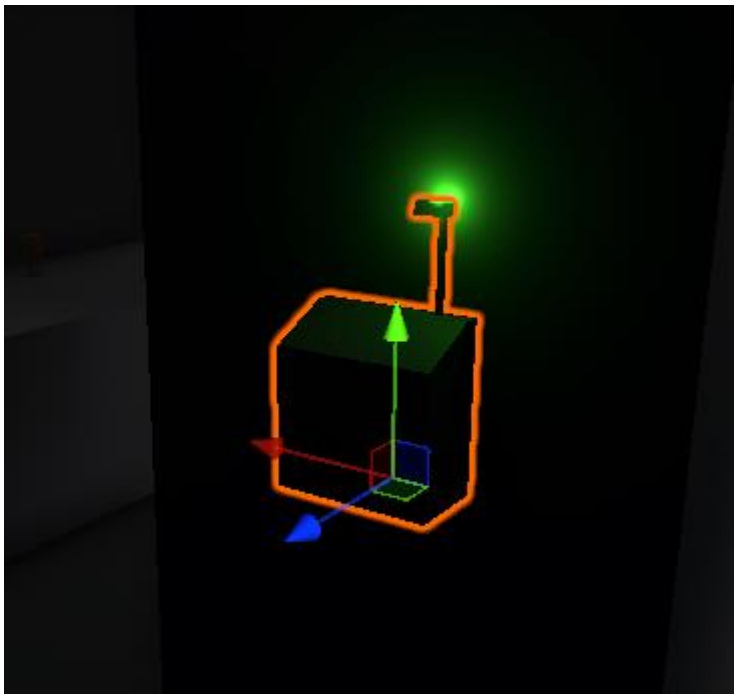


Figure 7.1.5- Wall phone that teaches player the dialogue system.

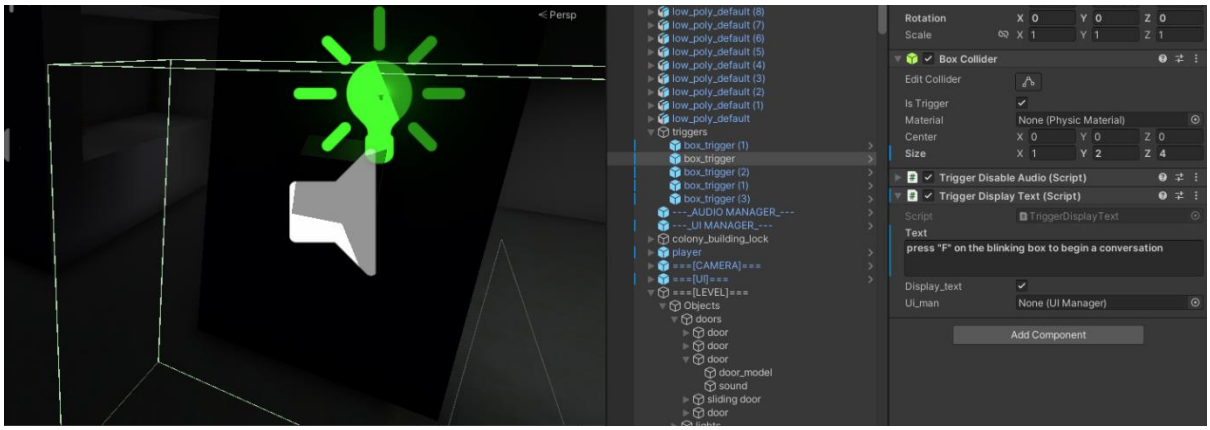


Figure 7.1.6 - Box trigger telling player to press "F" on the wall phone.

One of the smaller goals of creating this level was to not use UI markers that would show where the player needs to go. Not only is this such a small level that it can be traversed in roughly 10 minutes, but I also wanted to encourage exploration so telling the player the correct place to go outright defeated the point.

When the player leaves the apartment building, they can travel two ways to the police station. The first is on the right, indicated with a glowing barrel with the second indicated with a ceiling light above a door.

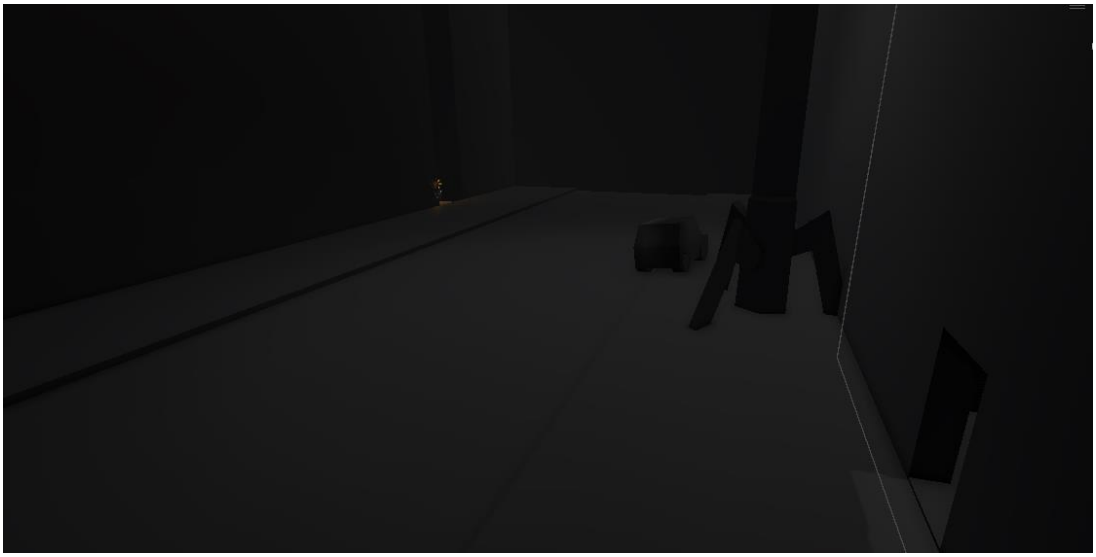


Figure 7.1.7 - Passage on the right of the apartment exit.

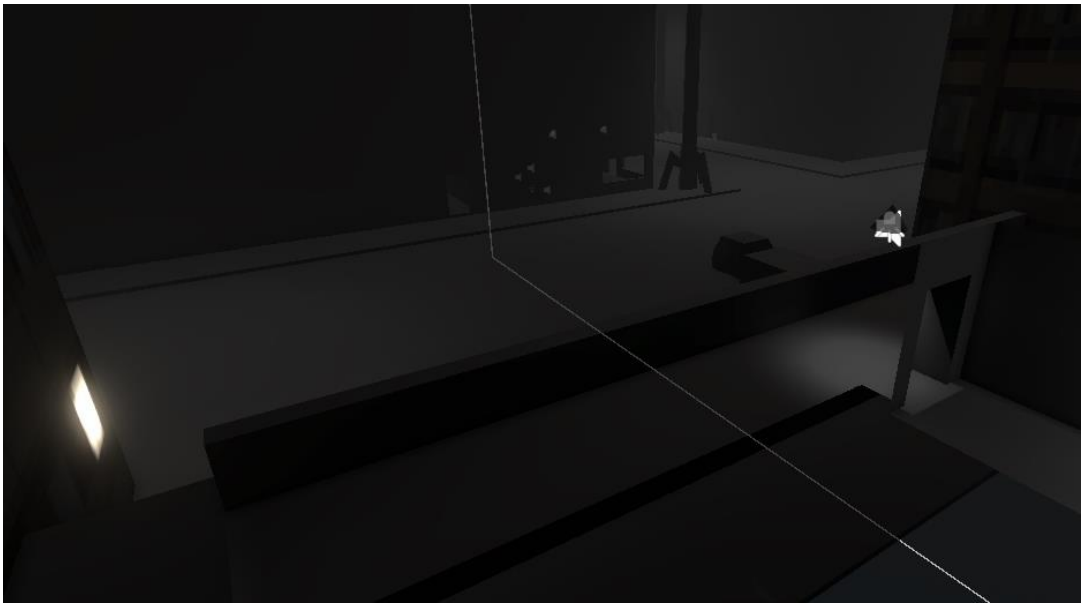


Figure 7.1.8 - Passage on the left of the apartment exit.

If the player chooses the right path, this will lead them down an alleyway and in front of the police station, indicated with the red and blue lights.

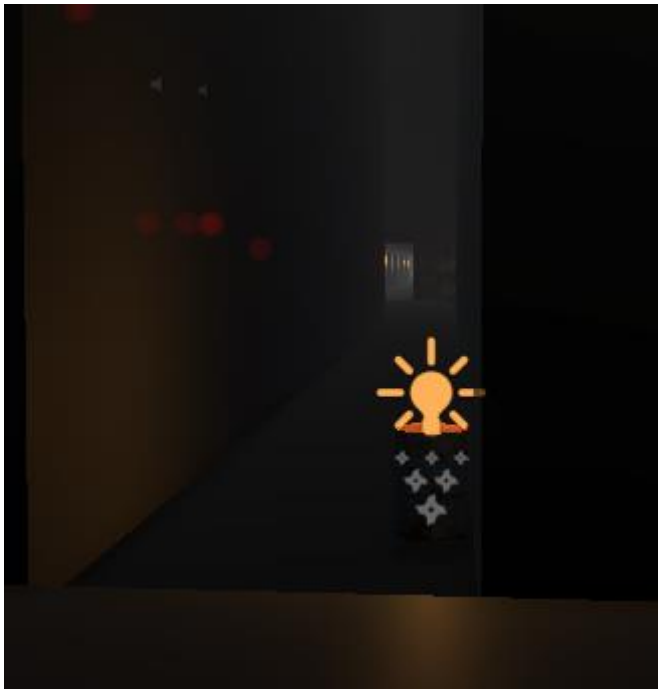


Figure 7.1.9 - Alleyway the player can travel through.

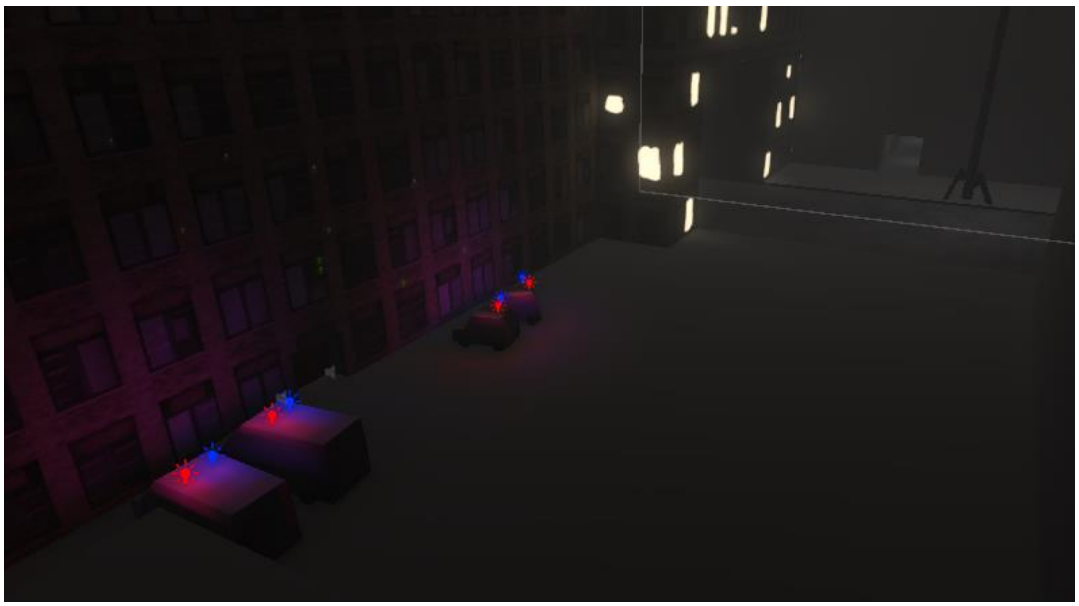


Figure 7.1.10 - Police station and cars in front.

If the player takes the left path, this will bring the player through a rundown apartment, up a maintenance shaft, through a dance hall, and eventually to the police station. This path has some additional items to make up for the additional journey length.

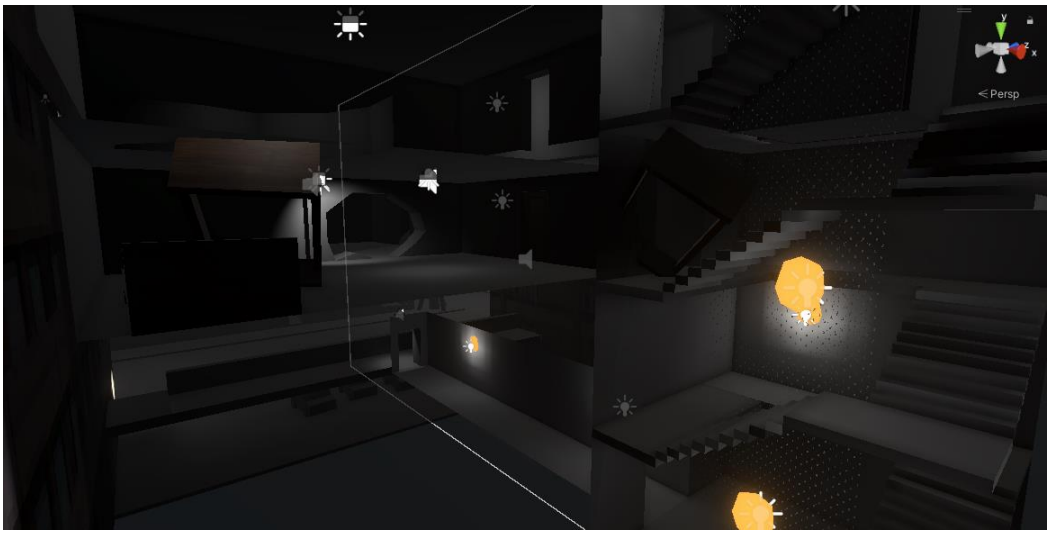


Figure 7.1.11 - Abandoned apartment.

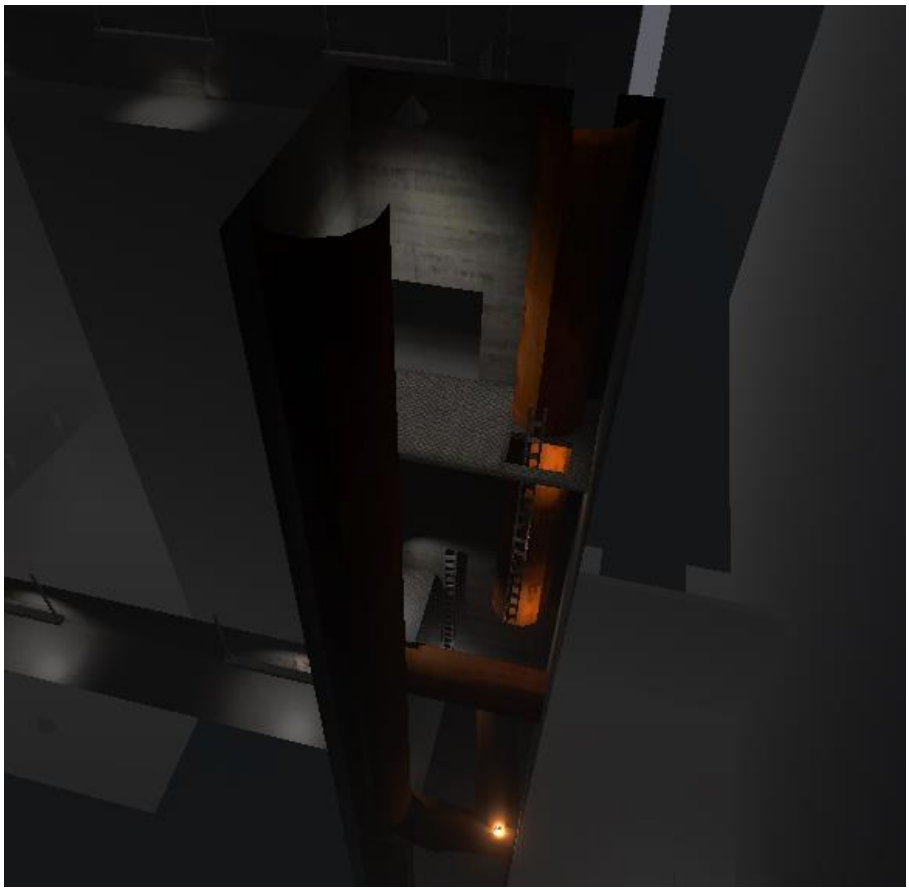


Figure 7.1.12 - Maintenance shaft.

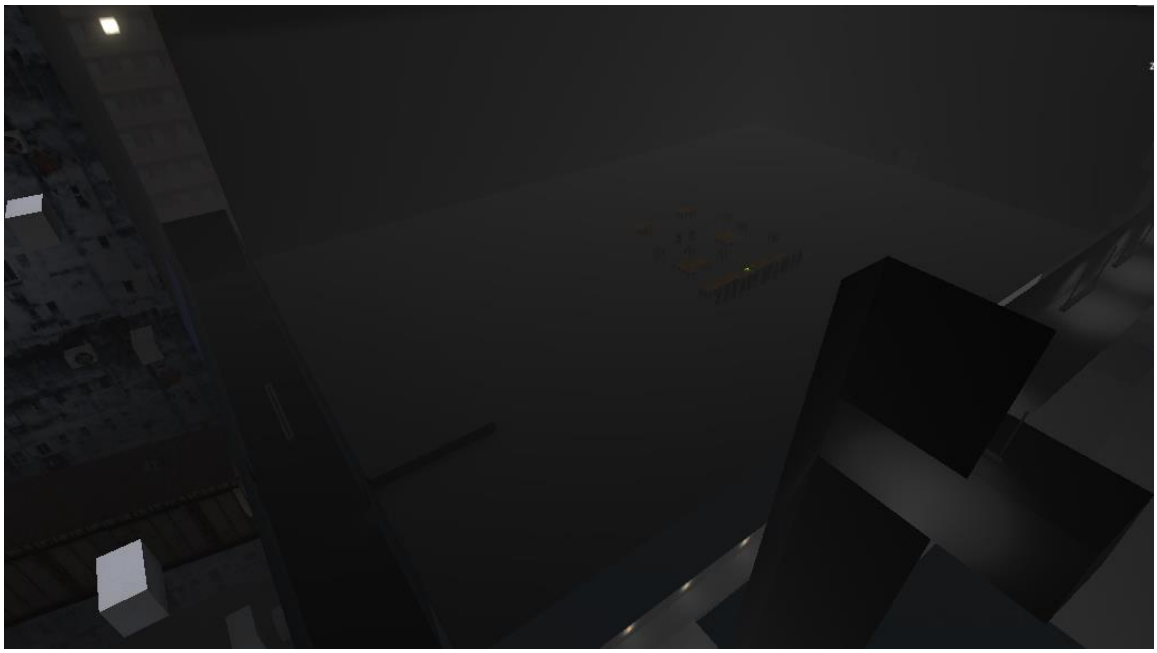


Figure 7.1.13 - Abandoned dance hall.

Figure 7.1.14 - Apartment complex linked to police station.

Figure 7.1.15 - Room above police station containing books and records.

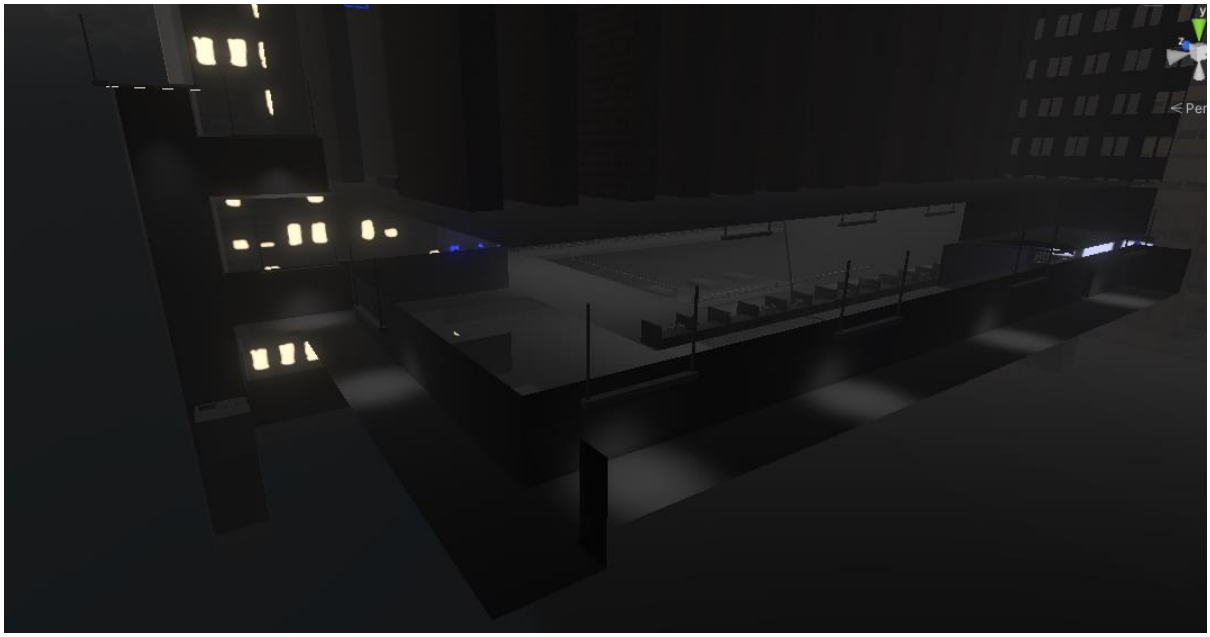


Figure 7.1.16 - Police station and hallway to police chief.

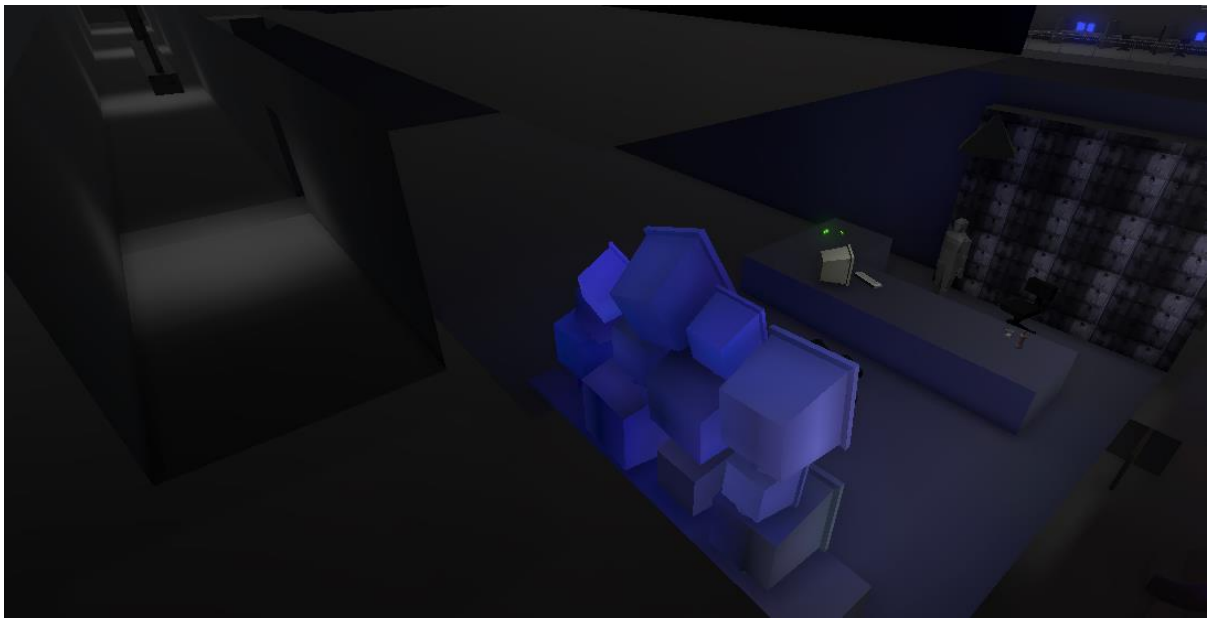


Figure 7.1.17 - Police chief office.

With the police station done, the next place was the building containing the scavs and their leader. When thinking of designs for this building, one idea was to create a building very similar to the Bradbury Building in Los Angles, America. This is a very famous building included in many famous films like *Blade Runner* and *Chinatown* (Wikipedia, 2023). It's also been included in some games like *E.Y.E: Divine Cybermancy* and *Peripeteia*, both of which influenced Project Deckard's direction.

Figure 7.1.18 - Outside of scav building.

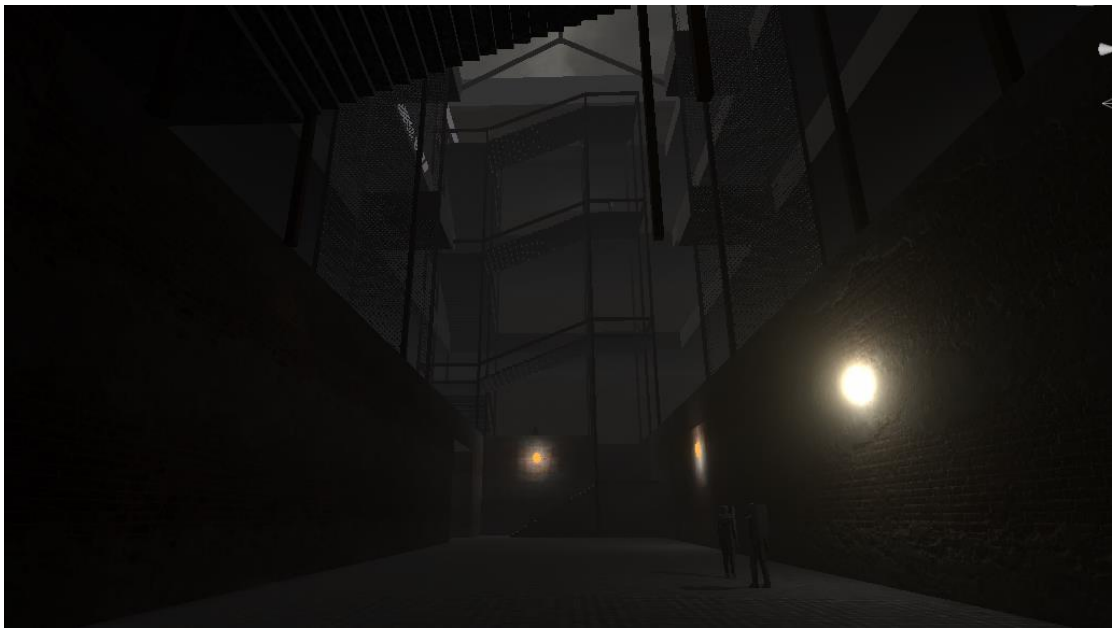


Figure 7.1.19 - Inside of scav building.

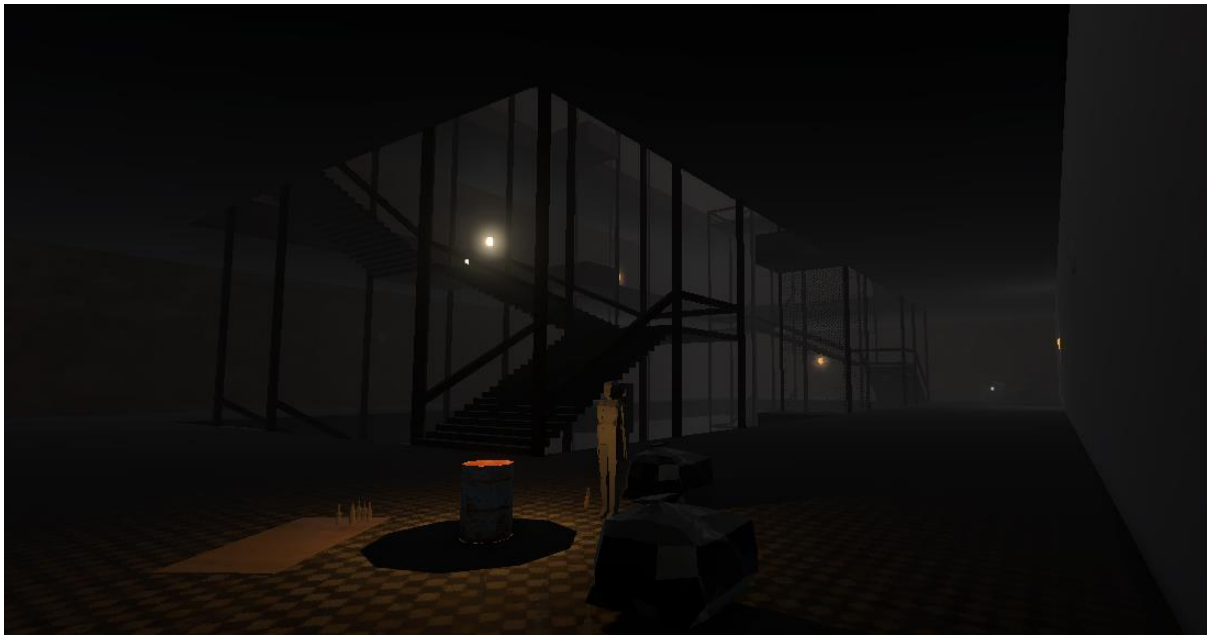
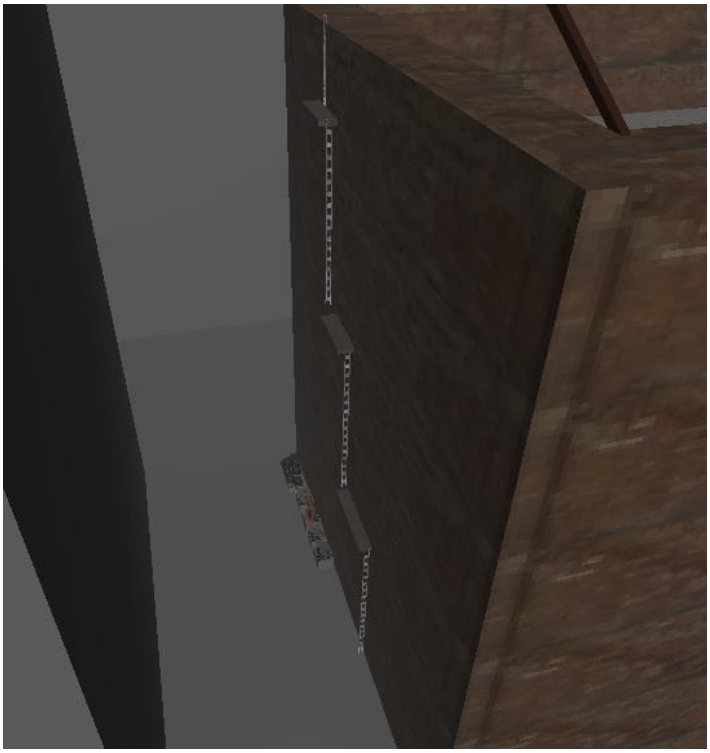


Figure 7.1.20 - Picture of the scav building and the Bradbury building for comparison (Wikimedia, 2022).

To give the player choice on how they want to enter the scav building, I created a ladder model and script that allows the player to climb their way to the top, although with some difficulty as the ladder script is very rudimentary.



I also added a vent with a light shining through it to grab the player's attention when exploring behind the building.

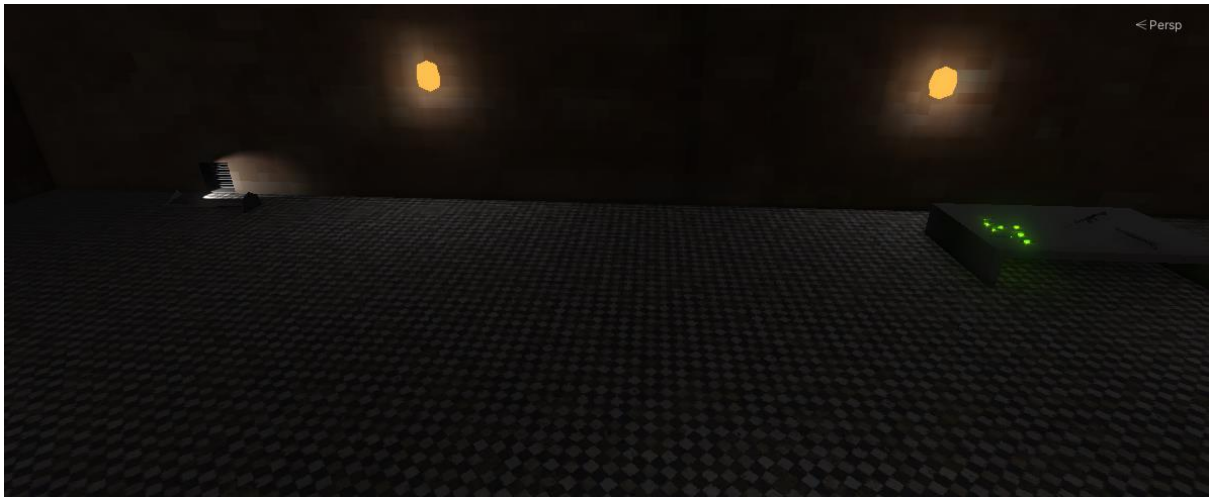
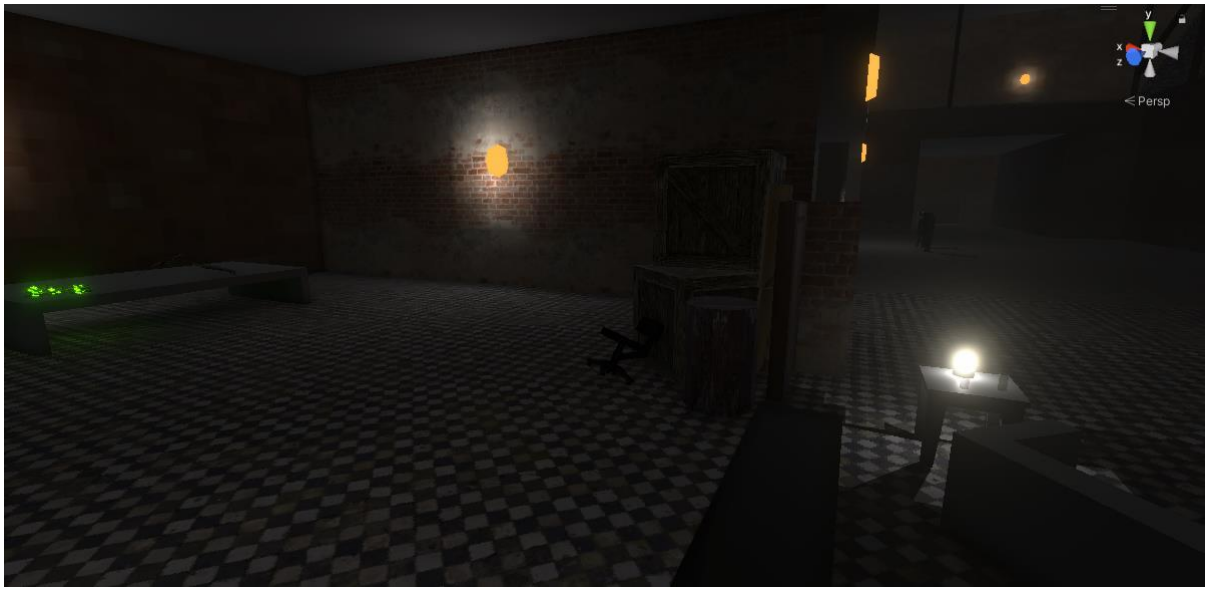


Figure 7.1.21 - Room behind vent containing two weapons and Medipens to regain health.



I created a small script to allow planks of plywood to stop doors opening until the health of the plywood has reached or is below “0”, further encouraging the player to explore the level.

If the player wishes to eliminate the scav leader because of his bounty, I also added another script that checks for fall damage on objects. If this object were to fall and the magnitude of its velocity were to exceed a threshold, it will destroy the original asset and instantiate a new “broken” version with physics, along with damaging anything it hit. I stacked a pile of cinderblocks on top of a plank of plywood above the scav leader, so now, if the player shoots the plank, it will shatter and the blocks will damage the scav leader, eliminating him without turning the rest of his gang aggressive.

I also added a small weapon pickup on a broken sky-bridge, if the player manages to see the flashing lamp from the scav building, or a flashing light from where the player can use a cable car to enter the skyscraper.

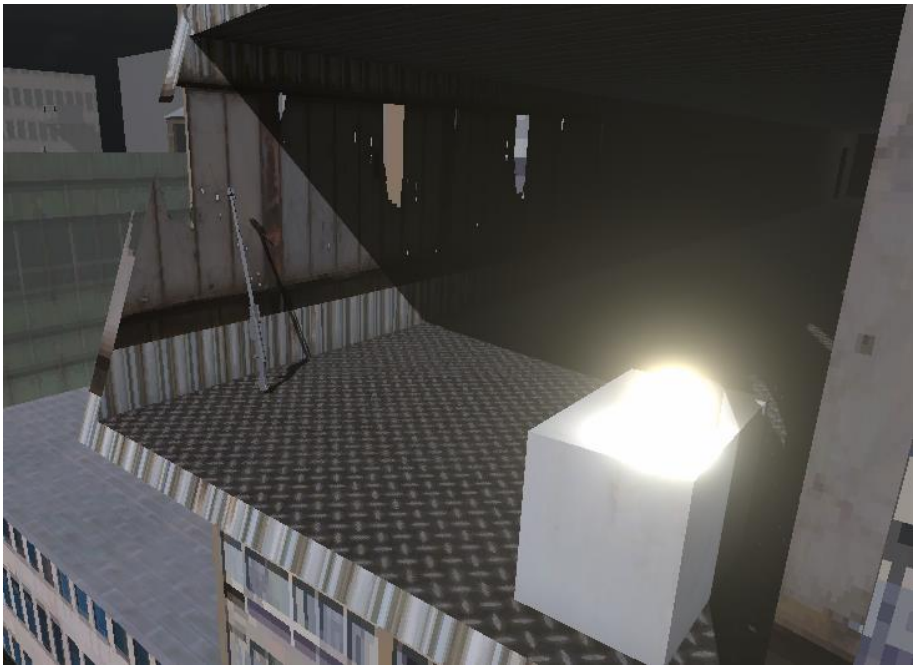


Figure 7.1.22 - Weapon drop and light.

Figure 7.1.23 - Player has line-of-sight to the scav leader.

Figure 7.1.24 - Lamp light can be seen from front entrance of the scav base.

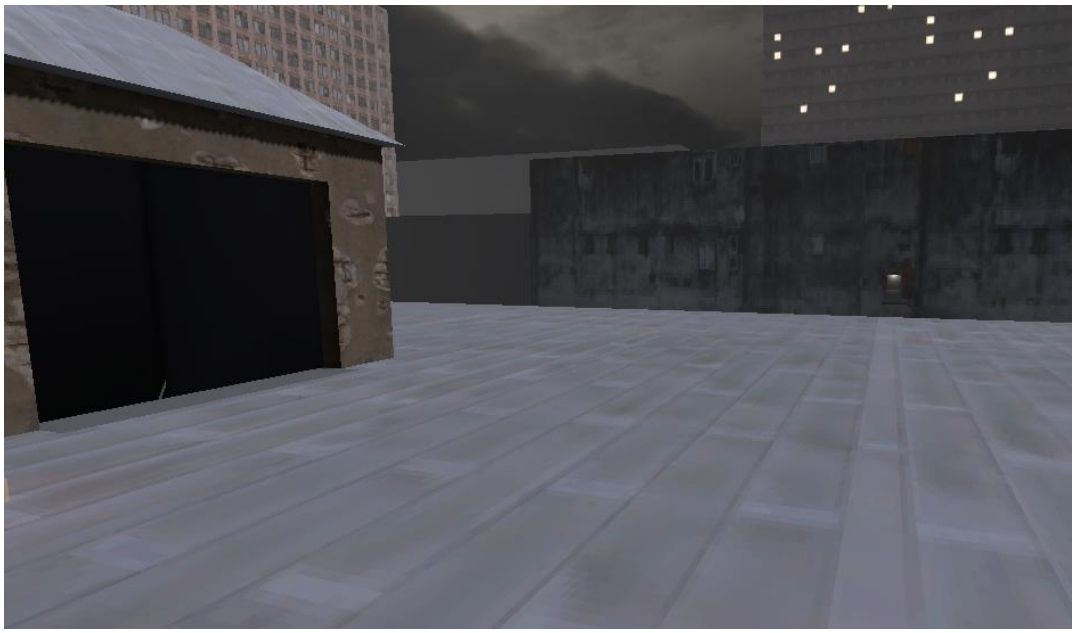


Figure 7.1.25 - Alternative way to find weapon drop from the cable car building.

Figure 7.1.26 - Cinder blocks above scav leader.

```
Unity Script (26 asset references) | 0 references
public class FallDamage : MonoBehaviour
{
    Vector3 init_vel, col_vel;

    public float fall_damage_treshold = 10f, final_vel;

    public Health health;


    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        if(GetComponent<Health>())
            health = GetComponent<Health>();
    }

    // Update is called once per frame
    Unity Message | 0 references
    void FixedUpdate()
    {
        init_vel = GetComponent<Rigidbody>().velocity;


    }

    Unity Message | 0 references
    private void OnCollisionEnter(Collision collision)
    {
        col_vel = GetComponent<Rigidbody>().velocity;
        final_vel = (col_vel - init_vel).magnitude;

        if (collision.gameObject.GetComponent<EntityHitbox>() && collision.gameObject.GetComponent<EntityHitbox>().enabled && final_vel >= fall_damage_treshold)
            collision.gameObject.GetComponentInParent<EntityHitbox>().OnRaycastHit(CalculateFallDamage(), transform.position, collision.rigidbody);

        else if (collision.gameObject.GetComponent<playerHitbox>() && collision.gameObject.GetComponent<playerHitbox>().enabled && final_vel >= fall_damage_treshold)
            collision.gameObject.GetComponent<playerHitbox>().onRaycastHitPlayer(CalculateFallDamage());

        if (final_vel >= fall_damage_treshold)
        {
            if (health != null)
                health.health -= CalculateFallDamage();
        }
    }

    3 references
    float CalculateFallDamage()
    {
        return final_vel * .3f * GetComponent<Rigidbody>().mass;
    }
```

Figure 7.1.27 - FallDamage script used for collision-based fall damage.

8 Arduino Controller

After doing some small tests with the Arduino controller, I was able to get the player to move on the x and z axis, as well as rotate the camera on the x and y axis. To move the player, I configured four buttons as inputs to simulate WASD keyboard controls for video games.

```
Unity Script (1 asset reference) | 0 references
public class UduinoPlayerRBMovement : MonoBehaviour
{
    public rigidbodyMovement rb_movement;

    private int w_button, s_button, a_button, d_button;


    int z_dir = 0, x_dir = 0;

    public float speed = 2f;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        UduinoManager.Instance.pinMode(11, PinMode.Input_pullup);
        UduinoManager.Instance.pinMode(10, PinMode.Input_pullup);
        UduinoManager.Instance.pinMode(9, PinMode.Input_pullup);
        UduinoManager.Instance.pinMode(8, PinMode.Input_pullup);
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        w_button = UduinoManager.Instance.digitalRead(11);
        s_button = UduinoManager.Instance.digitalRead(10);
        a_button = UduinoManager.Instance.digitalRead(9);
        d_button = UduinoManager.Instance.digitalRead(8);

        if(w_button>0)
            z_dir = 1;
        else if(s_button>0)
            z_dir = -1;
        else
            z_dir = 0;


        if (d_button > 0)
            x_dir = 1;
        else if (a_button > 0)
            x_dir = -1;
        else
            x_dir = 0;


        rb_movement.PlayerMovement(new Vector3(x_dir * speed, 0f, z_dir * speed));

        rb_movement.ExternalMovePlayer();
    }
}
```

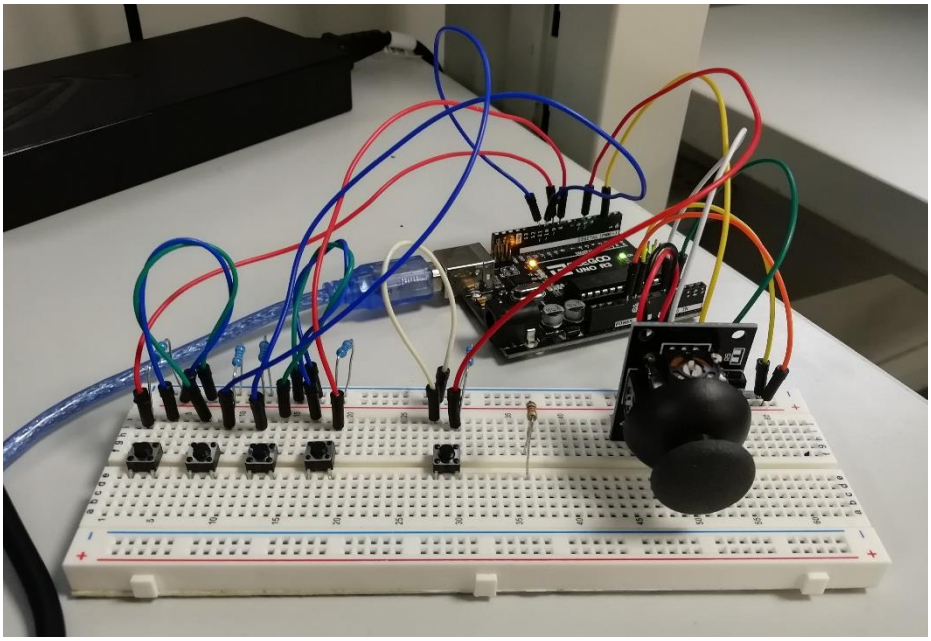
Pressing the *w_button* change the *z_dir* variable to “1”, causing the player to move forward, and pressing the *s_button* will change *z_dir* to “-1”, moving the player backwards. The strafe buttons, *a_button* and *d_button*, operate on the same principle.

To move the camera, we use the exact same method of moving the player, but instead the player camera will be rotated on the x and y axis. By using a PS2 joystick, a value will be outputted with relative ease to the user. The only issue encountered is the idle values of the joystick would fluctuate a small amount, ±1 unit for both the x and y axis. To mitigate this, I had the script only move the camera when the value being read was ±5 units above the base idle value, creating a "dead zone" where no input is read.

```
public class UduinoPlayerCameraMovement : MonoBehaviour
{
    public player_look p_l;
    public Camera player_camera;
    public Transform player_body;
    public Transform player_weapon;
    public int x_axis, y_axis;
    int base_x, base_y;

    float xRotation;
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        UduinoManager.Instance.pinMode(AnalogPin.A0, PinMode.Input);
        UduinoManager.Instance.pinMode(AnalogPin.A1, PinMode.Input);

        //  base values for joystick
        base_x = UduinoManager.Instance.analogRead(AnalogPin.A0);
        base_y = UduinoManager.Instance.analogRead(AnalogPin.A1);
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        x_axis = UduinoManager.Instance.analogRead(AnalogPin.A0);
        y_axis = UduinoManager.Instance.analogRead(AnalogPin.A1);
        float x_dir = 0f, y_dir = 0f;

        if (x_axis >= base_x + 5f)
            x_dir = 1f;
        else if (x_axis <= base_x - 5f)
            x_dir = -1f;
        else
            x_dir = 0f;


        if (y_axis >= base_y + 5f)
            y_dir = 1f;
        else if (y_axis <= base_y - 5f)
            y_dir = -1f;
        else
            y_dir = 0f;


        xRotation -= x_dir;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);
        player_camera.transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
        player_body.Rotate(Vector3.up * y_dir);
        player_weapon.localRotation = player_camera.transform.localRotation;
    }
```

Finally, to let the player be able to fight any enemies around the level, I needed to create another script that allowed the player to switch and fire weapons. The joystick has a button built in, so this was used to let the player cycle through their loadout and equip weapons available to them. Then to fire, a new button was added that when pressed, the gun they are holding fires (Wikimedia, 2022).

```
Unity Script (1 asset reference) | 0 references
public class UduinoPlayerWeapons : MonoBehaviour
{
    public PlayerInventory player_inv;
    public PlayerGun player_gun;

    bool pressed_weapon_switch_button = false;

    int i = 0;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        UduinoManager.Instance.pinMode(2, PinMode.Input_pullup);
        UduinoManager.Instance.pinMode(5, PinMode.Input_pullup);
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        if(UduinoManager.Instance.digitalRead(2) == 0 && !pressed_weapon_switch_button)
        {
            if (i > player_inv.loadout.Length)
                i = 0;

            player_inv.EquipWeapon(i);
            i++;

            pressed_weapon_switch_button = true;
        }

        if (UduinoManager.Instance.digitalRead(2) == 1)
            pressed_weapon_switch_button = false;

        if (UduinoManager.Instance.digitalRead(5) == 1 && player_gun.player_hot_key.equipted_item != null &&
            !player_gun.player_hot_key.equipted_item.is_melee_weapon && !player_gun.player_hot_key.equipted_item.is_throwable)
            player_gun.uduino_con_fire = true;
        else
            player_gun.uduino_con_fire = false;
    }
}
```



9. Conclusion

In conclusion, I am quite happy with Project Deckard as it achieved many of the design goals I had set out when planning its initial design. While features like multiple suspects and a much larger level were cut from the final product, after gaining some experience and perspective, I realized I had set the bar too high while having too little experience. Despite that, I was able to create systems that had more depth, bringing the player experience closer to a gaming experience I had envisioned when beginning Project Deckard's development.

During the development of Project Deckard, I learnt many new skills, such as game development, a new programming language, project management, along with model creation, rigging, and animation. These skills will be very useful and allow me to pursue a variety of careers, and further progress my future goals.

After enough time and experience, I'd like to revisit this project as it has a lot of potential to do well in a market that lacks gaming experiences like immersive simulations.

10. Poster

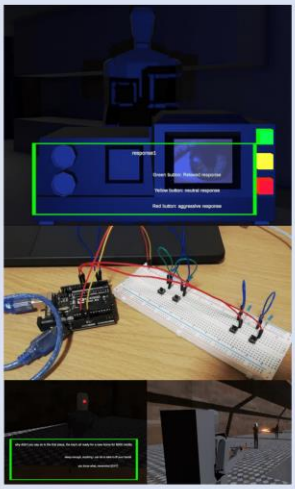
Project Deckard

Maghnus O Donnell – G00377514
BEng in Software and Electronic Engineering

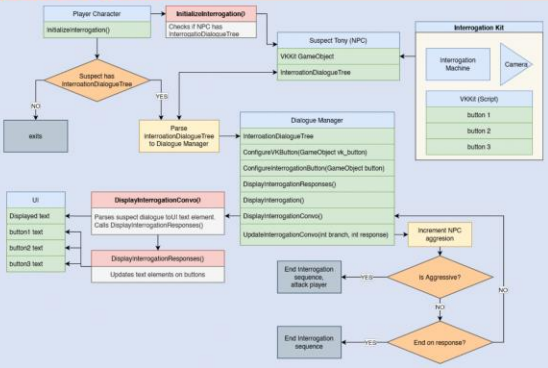
Ollscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

Project Overview

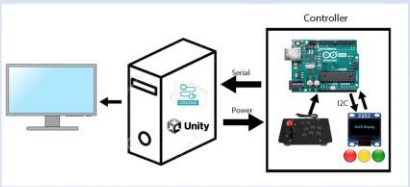
Project Deckard is a video game that has the player control a bounty hunter with an Arduino-based controller, to track down and interrogate an array of suspects to discover the randomly chosen target, however they deem fit.
The Arduino controller will have a variety of buttons and a joystick to help the player navigate the desolate city before them.
Project Deckard will pull design queues from games like Deus Ex, Half Life 2, and Prey 2017.



UML Diagram



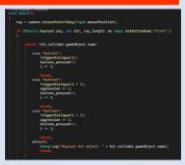
Block Diagram



Link to the game

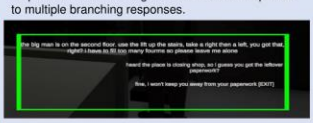
Results

The initial version of the dialogue script used a if statement, constantly polling for responses from a button press. Worked in the beginning but prone to bugs, along with poor scalability.



The current version now uses a binary-tree design, that displays dialogue when a button requests the Dialogue Manager, displaying the character's response at runtime. This improved reliability and scalability vastly, compared to the initial version.

Thanks to the binary tree format of the dialogue choices, there is no set limit to the number of dialogue responses. This can range from either one response to multiple branching responses.



Conclusion

While there are improvements to be applied to the AI state machines and game models, Project Deckard is now playable from start to finish with both mouse and keyboard, and the Arduino controller.

11 References

Isla, D. (2006). Retrieved from https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

Orkin, J. (2006). Retrieved from https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf

Unity. (2021). Retrieved from https://docs.unity3d.com/Manual/class-ScriptableObject.html

Wikimedia. (2022). Retrieved from https://en.wikipedia.org/wiki/Bradbury_Building#/media/File:Bradbury_Building,_interior,_ironwork.jpg

Wikipedia. (2023). Retrieved from https://en.wikipedia.org/wiki/Unity_(game_engine)

Wikipedia. (2023). Retrieved from https://en.wikipedia.org/wiki/Bradbury_Building