Girish Pillai
Bander Alahmadi
Ahmed Alsayat

Intrusion and Access Control Simulation

COSC 729
Spring 2015

Abstract

The proposed system's goal is to have a security access among all the rooms in the environment. Authentications of an authorized person by using a card access is not secured enough. In case of losing or steeling an authorized person's card, the access will grant to an unauthorized person. So, we proposed a simulated system using sensors along with card control to verify the person identity. The sensor will check the tall of the person based on a stored data. The combination of using both card access and sensors are much secured. And to identify intrusion detection, the proposed system detects the person first by the access card and then by the bio features, like height, weight and facial features.

GOAL AND OBJECTIVES

- To design a stimulation of an existing building.
- To show the geo location of the person.
- To show where the person is trying to access.
- To stimulate a view using a camera.
- To show the success and failure cases of access.

Introduction:

The security issues play a vital role at most of the life entities. One of these entities is the security buildings. It is very challenging to guarantee that you have a full secured environment. Especially when it goes to a sensitive and critical information such as bank, prison, etc.

It is noticeable that many companies and institutes use the card access on doors. We see this system is vulnerable since the card might be lost and used by un-authorized person. The sensor is proposed to add more secure access to the environment.

Using a 3D environment shows the system scenarios as in real. In our case, the avatar plays a person who has an identification to access certain rooms. This shows how the idea could be implemented before putting the system through and also we would be able to find the bottle necks of the existing system and each approach.

Usage

The proposed system could be used as a precursor for the deployment of a new system or to find the bottle necks in the existing system.

USE Keys “W”,”A”,”s” and “D” or the arrow keys to navigate the player.

Use Q and E to turn the player.

Methodology

We have our system proposed by Unity3D environment. The scene is crated inside the Unity as well as all objects. The scene typically is a one floor contains multiple rooms. Each room has an access point where the avatar must be authorized to enter. So, we have added ten avatars into the scene each with different customs. Each avatar has an identity to be distinguished in the scene. A name on each avatar is placed to show its characteristics when goes to access the authenticated door. The door shouldn’t open unless the avatar is authenticated by the sensor placed on the top of each door.

The avatars are animated to move smoothly inside the scene and interact with the other existed objects. We had added vases, doors, wall frames, trashes and chairs.

When the avatar is authenticated to access the door, welcoming sound message will play. Otherwise, the sound message would be that you are not authorized. In addition, we have added a sound of the door clicks.

Coding and Functionality Implementations

Unity 3d was used for implementation.

First is to create an empty game object and make it the game manager and attach the script to it.

It has a start method and a restart method. Create a new empty game object named Maze and attach the script to it. Turn it into a prefab by dragging it into a new Prefabs folder that we also create to hold it. Once that's done, get rid of the instance in the hierarchy. We have created a prefab so that we can spin out many references from it. Make it 20 by 20. We'll store the cells in a 2D array and create a new MazeCell script to represent the cells. We also need a cell prefab to instantiate.

We need a 3D visualization for our cells. Create a new game object named Maze Cell and add the MazeCell component to it. Then create a default quad object, make it a child of the cell and set its rotation to (90,0,0). That gives us a very simple floor tile that fills the cell's area. Turn the whole thing into a prefab, get rid of the instance, and give Maze a reference to it. Lets add a Generate method to Maze that will take care of constructing the maze contents. We start with creating our 2D array and simply filling the entire grid with new cells by means of a double for-loop. We put the creation of individual cells in its own method. We instantiate a new cell, put it in the array and give it a descriptive name. We also make it a child object of our maze and position it so that the entire grid is centered.

We now have to change GameManager so it starts the coroutine properly. Also, it is important to stop the coroutine when the game is restarted, because it might not have finished generating yet. As we only have to worry about one coroutine, we can take care of this by simply callingStopAllCoroutines. So yes, you can press space while a maze is still being generated and it will immediately start generating a new one.

To generate a real maze, we will be adding cells to our maze in a random way instead of using the double loop that we're using at this moment. So we will probably be using maze coordinates to figure out where we are at any given step. As we are operating in a 2D space, we need to use two integers. It would be convenient if we could manipulate the coordinates as a single value, like Vector2 but with ints instead of floats. Unfortunately such a structure does not exist, but we can create one ourselves.

Let's add a new IntVector2 script and make it a struct instead of a class. We give it a public x and z integer. That gives us two integers bundled together as a single value. We'll also add a special constructor method to it, which allows us to define values via new IntVector2(1, 2).

We will most likely be adding these vectors together at some point. We could create a method for that. But it would be even more convenient if we could simply use the + operator. Fortunately, we can do this by creating an operator method, which is how Unity's vectors support operation as well. So yes, adding two vectors means that you're calling a method.

Let's add support for the + operator now. You can define the other operators as well, but addition is all we need here.

Let's do away with our double loop that Maze uses to generate a regular pattern of cells. Instead we'll pick some random coordinates inside the maze and start generating a line of cells from there, until we run out of the room.

To make this work we have to also add a RandomCoordinates property to Maze that produces some coordinates inside it, plus a ContainsCoordinates method that checks whether some coordinates fall inside the maze. Let's make them public as they would be useful for anything that deals with rooms.

While we now tend to generate longer paths of cells, it's still far from a complete maze. We should really be smart about how we move from cell to cell.

It's time to keep track of the connections between cells. Each cell has four edges, each of which connects to a neighboring cell, unless it would lead outside of the maze. We could either create a a single bidirectional edge between two cells, or give each their own unidirectional edge. We choose the latter approach, because it is more flexible.

Add a script for the new MazeCellEdge component type. Give it a reference to the cell it belongs to and one to the other cell that it connects with. Also give it a direction so we remember its orientation.

Doors are another interesting element to add to our maze. Let's add a MazeDoor component that extends MazePassage. Because it will have a rotating part, add a public Transform variable to it named hinge.

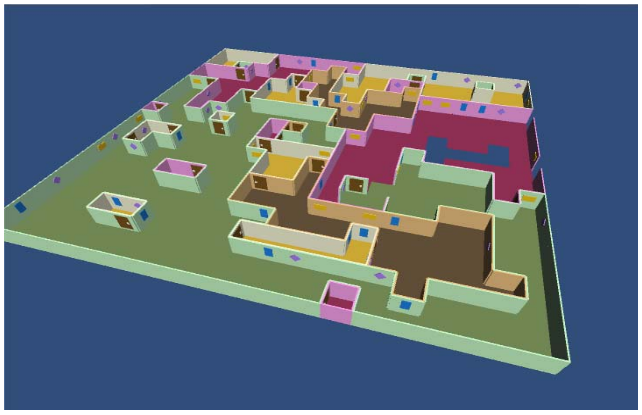
We will build a door frame from four cubes and put another cube in it with a new Door material, plus a door handle on the right side of it. To allow the door to rotate properly, add an empty game object named Hinge on the left side of the door with a Z-position of 0.5. Make the door and handle objects children of it. Add the MazeDoor component to the root object and connect its hinge, then turn it into a prefab.

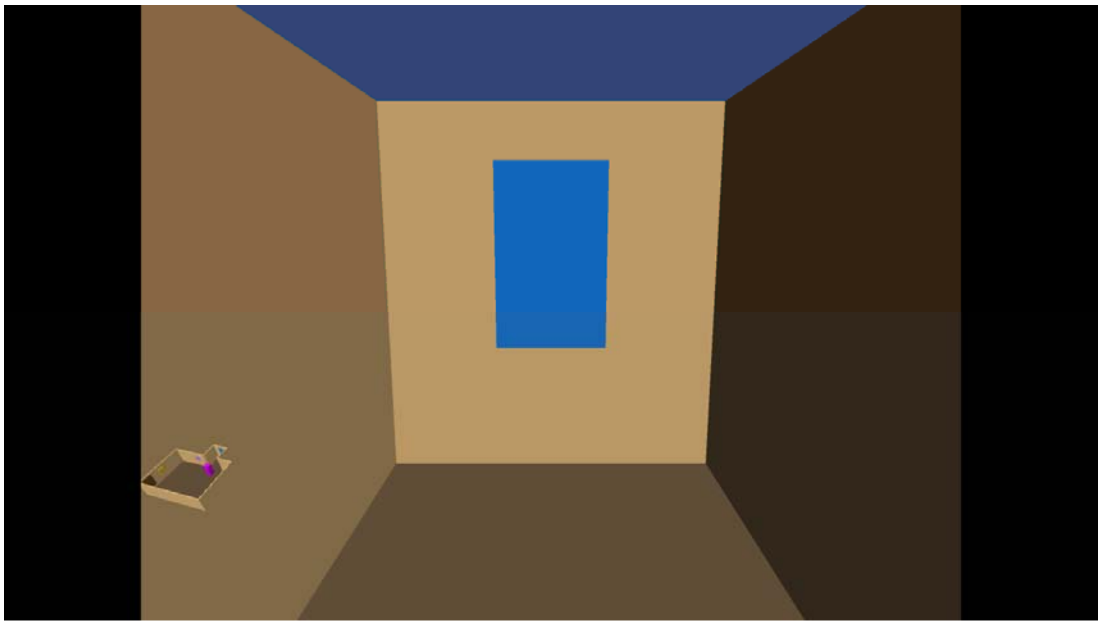
Adding Player

It's high time we walked around in our own maze. Create a simple player model, attach a newPlayer component that we create as well, and turn it into a prefab. Give Player a public method so we can tell it what cell it's in. Also give it an Update method that moves the player when an arrow key is pressed. Movement should only happen if the edge we would cross is a passage, otherwise we're blocked.

Coding and Functionality Implementations:

The Scene:





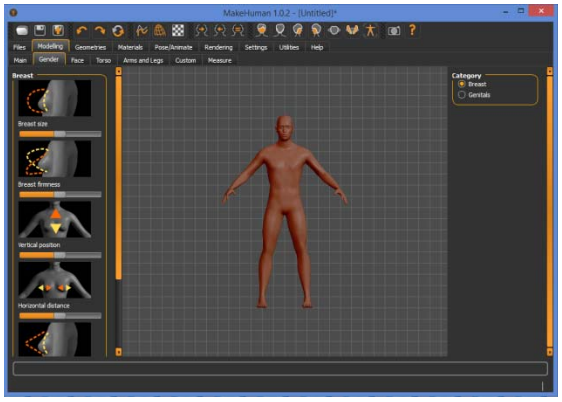
The movement inside the Scence: “GetKey Function”

```
private void Update () {
                        if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow)) {
                                Move(currentDirection);
                        }
                        else if (Input.GetKeyDown(KeyCode.D) || Input.GetKeyDown(KeyCode.RightArrow)) {
                                Move(currentDirection.GetNextClockwise());
                        }
```

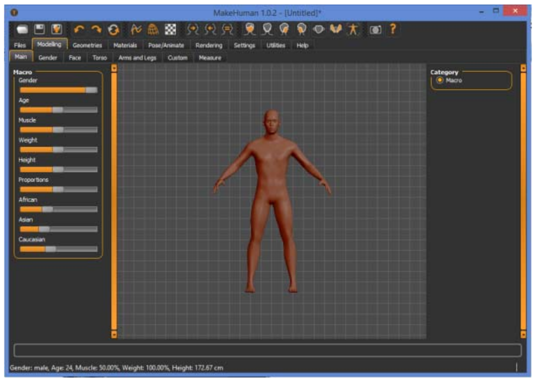
```
else if (Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.DownArrow)) {
		Move(currentDirection.GetOpposite());
}
else if (Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.LeftArrow)) {
		Move(currentDirection.GetNextCounterclockwise());
}
else if (Input.GetKeyDown(KeyCode.Q)) {
		Look(currentDirection.GetNextCounterclockwise());
}
else if (Input.GetKeyDown(KeyCode.E)) {
		Look(currentDirection.GetNextClockwise());
}
```

Avatar Creation:

we create the avatars using MakeHuman software which enable users to create human and controlling the human shape and clothes and every single detail.

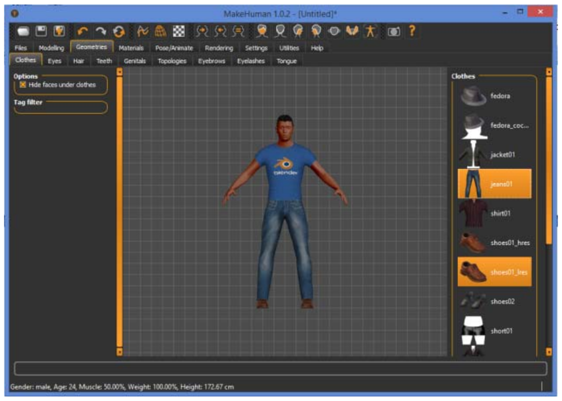


The user also can control the ethnicity of the human.



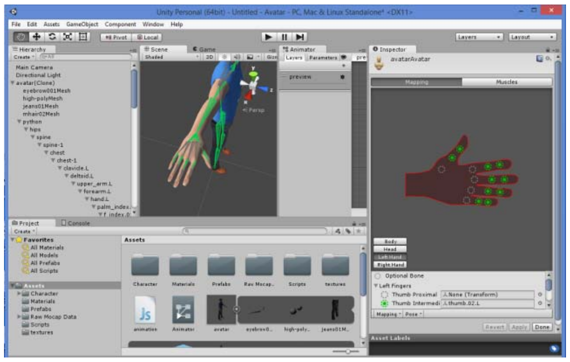
Gender: male, Age: 24, Muscle: 50.00%, Weight: 100.00%, Height: 172.67 cm

The user also can control the clothes and the hair of the avatar.

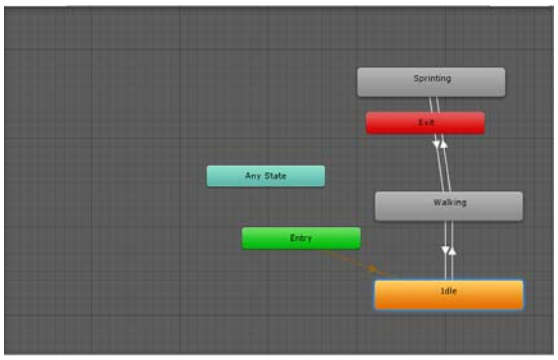


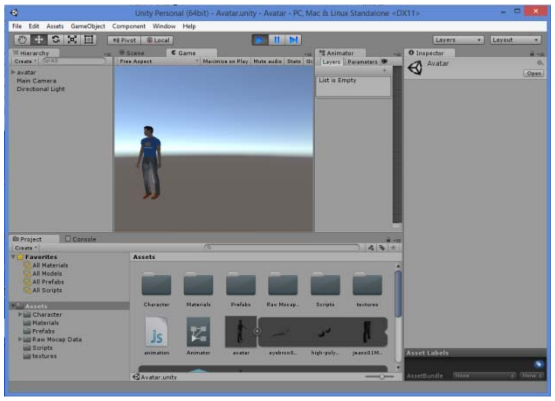
After finishing creating the avatar, we then export the avatar as filmbox (.fbx) file.

Then we import that file into Unity3D. When we import the avatar into Unity3D, we need to add the texture to the avatar. Also we need to fix the avatar hand.



The next step is adding motion to that avatar. We use Raw Mocap animation from the Assets store. We create new animator that has three states: Idle, Walking, and Sprinting. Each state has one motion from Raw Mocap motions.





```
var animator : Animator; //stores the animator component
var v : float; //vertical movements
var h : float; //horizontal movements
var sprint : float;

function Start () {

animator = GetComponent(Animator); //assigns Animator component when we start the game

}

function Update () {

v = Input.GetAxis("Vertical");
h = Input.GetAxis("Horizontal");
Sprinting();

}

function FixedUpdate () {

//set the "Walk" parameter to the v axis value
animator.SetFloat ("Walk", v);
animator.SetFloat ("Turn", h);
animator.SetFloat("Sprint", sprint);

}

function Sprinting () {
if(Input.GetButton("Fire1")) {
sprint = 0.2;
}
else {

sprint = 0.0;
}

}
```

Work Distribution:

Girish Pillai__ Integrated the whole code and run it,doors,walkways, Testing ,documentation. Report contribution and slides.

Bander Alahmadi__ Avatar Creation Section, Report contribution and slides, Testing ,documentation.

Ahmed Alsayat__ Scene Creation, Movement inside the environment,Use getkeycode to trap values for user and camera movements, Testing ,documentation. Report and Slides Creation.

Future Work:

We may add a baseline data for each avatar. The data will contain a time the avatar uses to slide his card into the access point when the sensor capture the body. We also may use the way of opening the door (the angle when the door is opened by avatar) as a baseline database. If the angle ratio doesn't meet the data stored, a security alarm will be triggered.