Hao WAN (hxw161730)

Final Project Report CS6331 Multimedia System 2017 Fall

3D Robot Control in a 3D Scene Generated from RGB-D Image

Contents:

1. Description of the project

In this project, the problem I'm dealing with is to use a RGB-D dataset to visualize a 3D scene, and then render a robot model in this scene, control the robot to make it move to a marker position. The robot will be made animated and will be with sound effects when moving.

2. Summary of the Implementations

Firstly, to reconstruct the 3D scene, we use OpenCV and Point Cloud Library to convert the RGB-D image into a point cloud dataset. We choose an appropriate format and import it in Unity. Then we construct a 3D scene with this point cloud file and render an animated robot model in it. To make the robot model move to a marker position in the 3D scene, we add some C# code in the corresponding character controller script. We also write a few scripts to implement sound effects when the character walks or runs.

In this project, a lot of tools, libraries and steps are involved. The following report is therefore divided into multiple sections. In each section, we only tackle with one step. We go through step by step from section 3 to section 14, in the implementing order, to explain how we realize the whole project. Section 15 and 16 are for referencing and indexing the attached files and scripts.

3. Get an RGB-D dataset

The RGB-D dataset is given. It is captured from the Kinect device in the Multimedia Lab. The given RGB-D dataset consists of two kinds of image: the normal color image, in JPG format; and the depth image, in PNG format.

The following is the RGB-D dataset we use for this project:

The color image (given by Multimedia Lab), color.jpg:

The depth image (given by Multimedia Lab), depth.png:

The one with the depth information shows how deep is every pixel in the color image to the sensor.

A long with the above dataset, an intrinsic matrix of the Kinect camera is also given as the following:

1036.1	*5.2187*	*953.2*
0	*1038.4*	*536.71*
0	*0*	*1*

A “Pinhole Camera Model” is introduced, with the corresponding virtual camera’s geometry. To understand the model, we browse this webpage: http://ksimek.github.io/2013/08/13/intrinsic/ . The intrinsic matrix indicates the intrinsic parameters of the camera. It transforms 3D camera coordinates to 2D homogeneous image coordinates. So, to reconstruct the 3D scene form the 2D image, we will also need this intrinsic matrix.

To straightly get the corresponding intrinsic parameters of the camera that we use to, we turn to the “Hartley and Zisserman” form of intrinsic matrix:

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

The form is from the webpage above. fx and fy are Focal Lengths, x0 and y0 are Principal Point Offsets, s is an Axis Skew parameter and is usually taken as 0.

So, in our case, we have: fx = 1036.1, fy = 1038.4, x0 = 953.2, y0 = 536.71, and s = 5.2187. Since s is too small comparing with the values of other parameters, we take it as 0. These parameters are for later use to reconstruct the 3D point cloud.

4. Convert into Point Cloud

This is the step from 2D image to 3D point cloud. The ideas and main script for implementing this step come from part II of this tutorial: https://github.com/gaoxiang12/rgbd-slam-tutorial-gx .

A point cloud is a dataset containing points in 3D coordinate system, and usually with color information at each point. The points in a point cloud dataset are always located on the external surfaces of the visible objects. A point cloud is easy to edit, display and filter because the points in it are individual and unrelated to each other.

To obtain a point cloud file from an RGB-D image, we need OpenCV and Point Cloud Library. OpenCV is used to read and take the information of every pixel from the color and depth image, and store them into matrices for later implementations. PCL has many useful tools so that we can create and save the point cloud dataset. The computing procedure from 2D to 3D is realized by a piece of C++ script.

For implementing this, firstly we need to build an environment that support OpenCV and PCL. We choose Ubuntu to build and run the libraries and the C++ script. It's a very friendly environment to install and integrate all the dependencies we need. It's also convenient to run the C++ code using CMake. I'm using Windows, so I get an Oracle VirtualBox first, and build an Ubuntu system on it.

According to the pinhole camera model, we have this formula:

(from the tutorial mentioned above)

$$u = \frac{x \cdot f_x}{z} + c_x$$

$$v = \frac{y \cdot f_y}{z} + c_y$$

$$d = z \cdot s$$

This formula shows that a 3D point (x, y, z) has a coordinate of (u, v) as a pixel in the 2D color image, with a depth parameter d indicated by the depth image. This is how we transform a 3D point to a 2D image, using the intrinsic matrix K mentioned in section 3.

But the s here is not the Axis Skew, it's a camera factor, or exactly the scaling factor designating a ratio of the depth give in the depth image over the real distance from the object to the sensor. It's a trivial factor in our project, because it doesn't affect the inside structure of the point cloud and we can do the scaling as we want in Unity later. This factor usually takes a value of 1000, 3000 or 5000, because we usually use mm as the unit in the depth image while for the real distance we use meter. So, in our project, we simply take 5000 to do the calculation.

According to the formula above, to reconstruct a 3D point cloud file from 2D image, we can write:

$$z = d/s$$

$$x = (u - c_x) \cdot z/f_x$$

$$y = (v - c_y) \cdot z/f_y$$

This is how we calculate the coordinate of a point in the point cloud.

We use the convert.cpp script to implement the calculation. The code is from the tutorial mentioned above, with some optimizations for this project. (Please refer to the "resources" folder to see the entire code.)

In the convert.cpp file, firstly, we include the OpenCV and PCL libraries, define a 3D point with color information and a point cloud structure using PCL. For the intrinsic parameters of the camera, we set them as the following:

```
const double camera_factor = 5000.0;
const double camera_cx = 953.2;
const double camera_cy = 536.71;
const double camera_fx = 1036.1;
const double camera_fy = 1038.4;
```

fx and fy are the focal lengths, cx and cy are the principal point offsets. We have defined them in the last step. For the camera factor or the scaling factor, we take 5000.

Then, in the main function, we use OpenCV to read every pixel of the color image and the depth image, and store them separately in matrix "Mat rgb" and "Mat depth". We create a new point cloud using PCL, too. Next, we implement a nested for loop to traverse every pixel of the depth image (m rows and n columns); depending on the formula above, we calculate the 3D coordinate x, y, z of every point in the depth image. At the same time, we get the corresponding color information from the matrix "Mat rgb" for the point. At last, we add this point to the point cloud and save it in a PLY file.

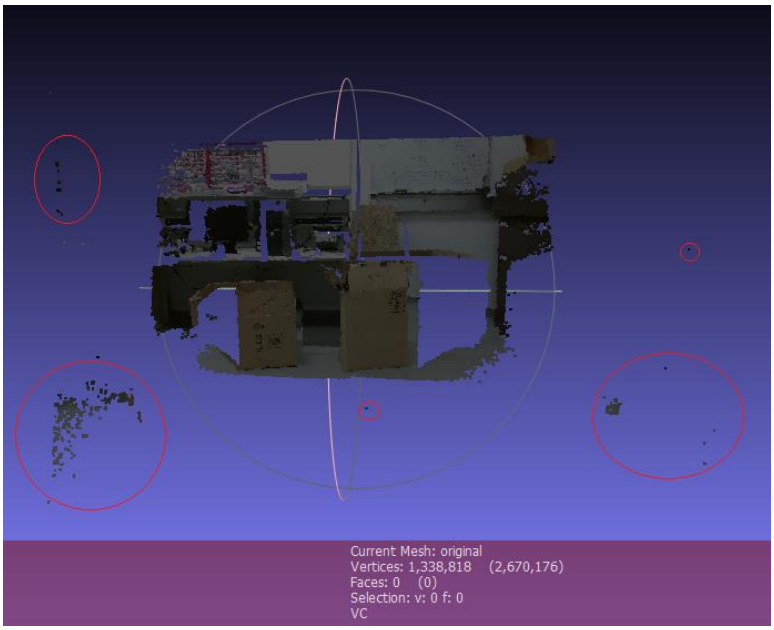
5. Optimize and Choose a final format

During the implementation, several optimizations are made. One of them is that when the original point cloud is obtained, I find that a few points are out of range.

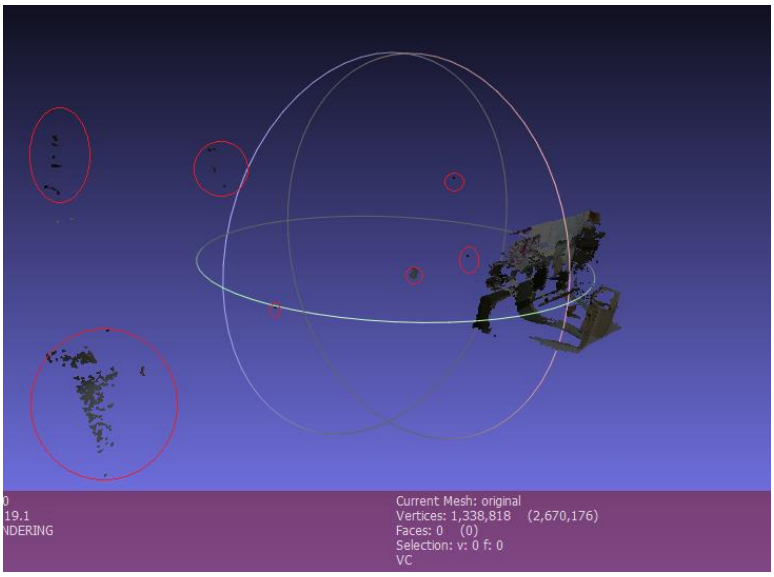
As the following pictures shows:

The red circles show these abnormal points with extremely huge values of depth.

The second picture is another view from the left side. The problem is more obvious under this view.



Another view of the problem:

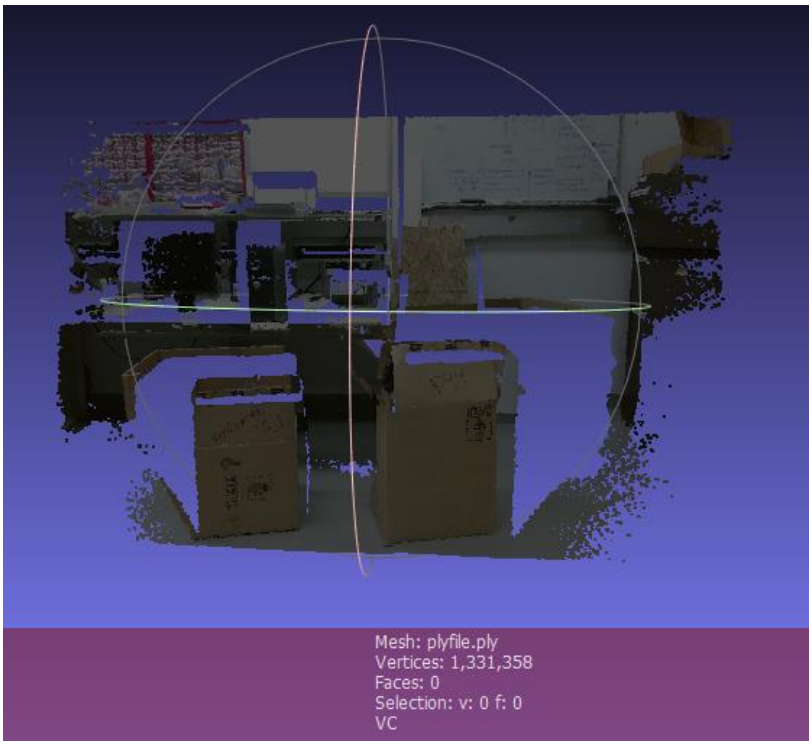


Normally, all the 3D points should stay within a range and build up the surface of the object. In this case, we can clearly tell that a few points have a very huge and wrong depth information. This problem may be caused by the incertitude of the sensor. To solve the problem and don't make difficulties when later utilizing the point cloud in Unity, I add a threshold to filter out these points. The original code is that if the depth information is 0 which means no meaningful value, we will simply skip it. And in our case, we should add that, when d is extremely large, like greater than a threshold, we should skip it, too. I test several values of threshold and make it 5000.

Implementing a threshold in the conver.cpp:

```
//if d has no meaningful value, skip this point
if (d == 0 || d > threshold)
    continue;
//or add a point to the point cloud
PointT p;
```

Then, after adding this threshold and filtering out the abnormally distant points, we have a following point cloud:



We lose about 7000 points but the whole cloud seems neater.

The above pictures are shown in a software named MeshLab using PLY point cloud files. When talking about the file formats of point cloud, we usually come to two kinds of formats: .PCD file and .PLY file. PCD stands for Point Cloud Data and PLY stands for Polygon File Format. PCD file contains the x, y, z coordinates and the rgba information for every 3D point. The color information is encoded as one uint32 value in PCD file. While for a PLY file, the coordinates are stored in form of vertex, face and edge lists, the rgb information is not encoded but directly stored with values between 0-255.

To view the point cloud file, we have 2 methods:

1) Use the software MeshLab

MeshLab is very convenient to use, but it doesn't support PCD file. Luckily it reads PLY file. So in the converting step we directly save the file as a .PLY file using ::io::savePLYFileBinary().

I prefer this method because later we will use MeshLab again to convert the file into another format .OFF to make it easy when importing into Unity.

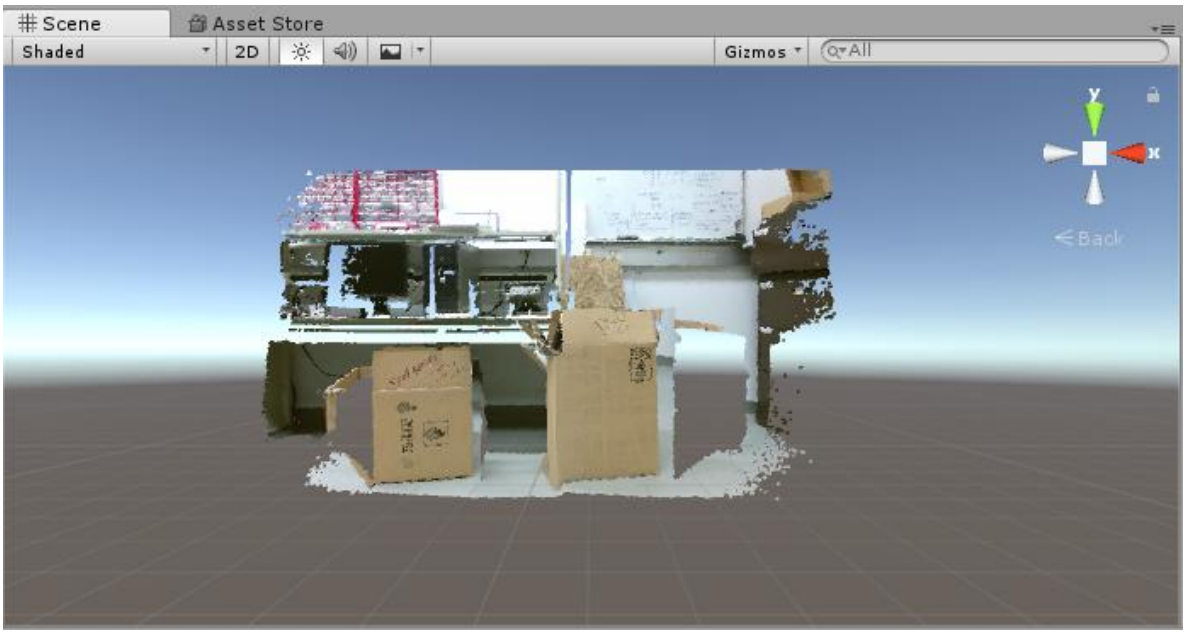
2) Use PCL tool pcl_viewer

pcl_viewer only takes PCD file. We use ::io::savePCDFile() to get a .PCD file.

The last thing to do is to convert the .PLY file into .OFF file using MeshLab, so that we can easily import the scene into Unity. The conversion is very easy using the UI of the software.

6. Import into Unity

In Unity, we can download a free asset named Point Cloud Free Viewer. It can render a point cloud file with .OFF format. The link of this asset is https://www.assetstore.unity3d.com/en/#!/content/19811 . We create a new project folder, and import the asset into it, follow the instructions, and there will be a "pointcloudfile" prefab in the project folder. The whole cloud is very large, containing more than 1 million points, so the prefab file in Unity is actually composed of 21 point cloud sub-files. We simple drag it into our scene. And then the point cloud is imported. As the following picture shows:

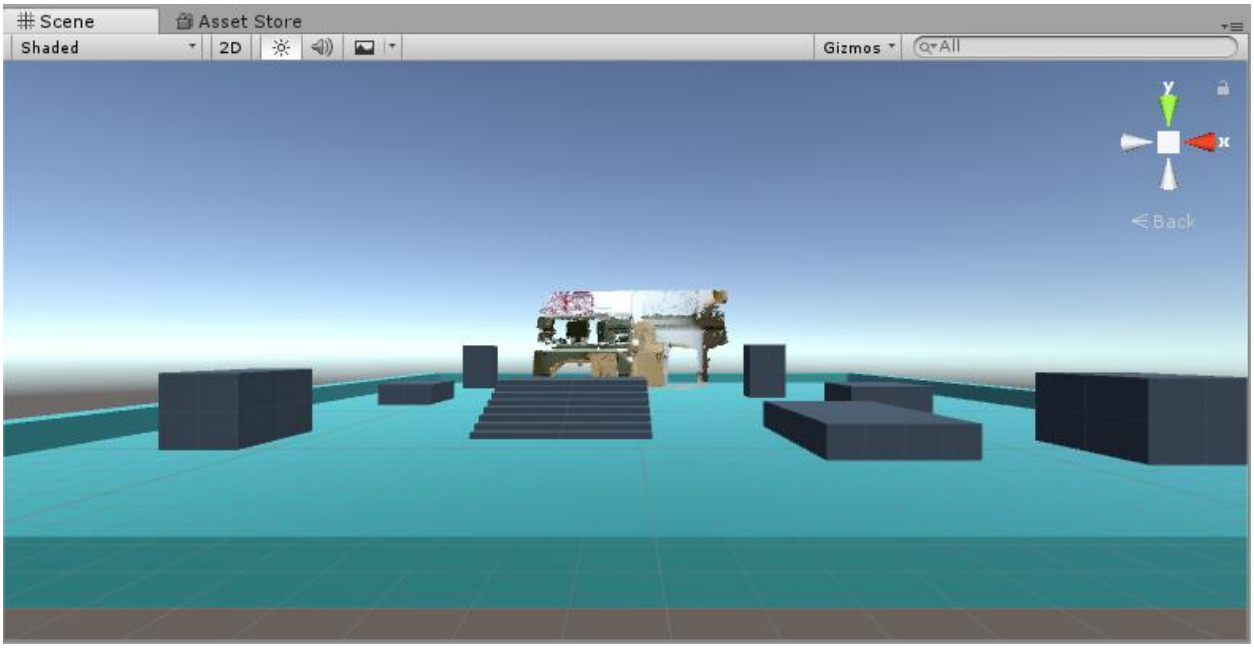


7. Build a simple scene in Unity

To start and get to know the basic operations in Unity, I finish the official Unity tutorial "roll a ball": https://unity3d.com/learn/tutorials/s/roll-ball-tutorial . After that, I'm able to create a ground plane with several walls and cube objects on it. I drag the point cloud into the scene, too. I also put a stair and ramp in the scene, by using the built-in Standard Assets. The following picture shows the very basic

scene with the point cloud we construct during the previous steps in Unity. For the next steps, we will bring a 3D robot model into the scene and make it walk around.

A 3D scene built in Unity, with the point cloud in it:



8. Import a robot model

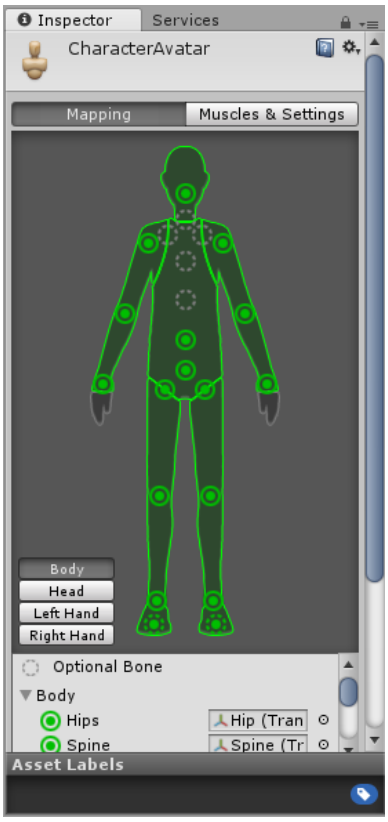
In this and the next step, I mainly follow a tutorial on Youtube to import a robot character and make it animated. I adopt some of the scripts in the tutorial. But this tutorial only teaches the basic functionalities with simple keyboard control. For implementing the features like click to move, showing an indicator or adding sound in the later steps, I turn to the official document of Unity and make a lot of modifications on the original scripts of this tutorial.

The link of the tutorial is: https://www.youtube.com/playlist?list=PLFt_AvWsXl0djuNM22htmz3BUtHHtOh7v . It's a very nice tutorial containing 9 short videos. The first 6 videos explain how to build a character model with walking and running animation from scratch in the software Blender. The last 3 videos deal with how to do basic control of the character in Unity.

I'm very interested in the character creation process with Blender. For the future update, I'd like very much to create my own character model using Blender, following the tutorial. But the time left for doing this project is limited, so, for the moment, I directly import the already-built character model into my project in Unity. It's a .blend file of a 3D robot character. The basic walking and running animation is included, too. Here is the link of the file: https://github.com/SebLague/Blender-to-Unity-Character-Creation/tree/master/Blend%20files (The last one is the file I use.)

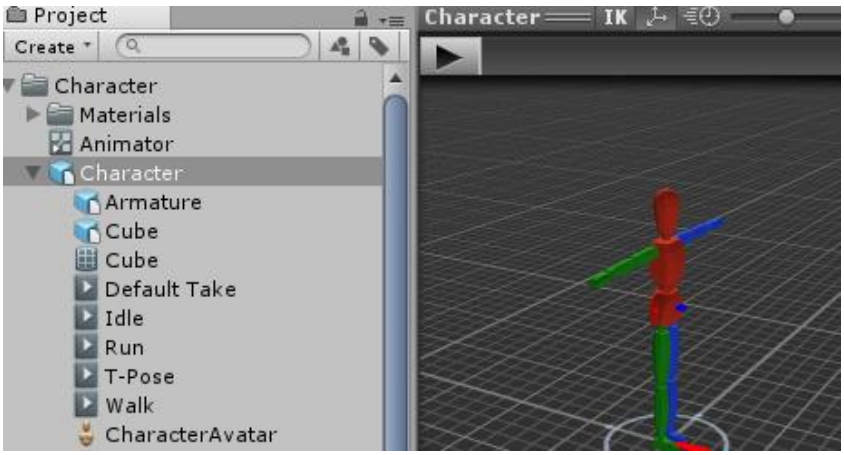
I simply drag the blend file into the Asset tab of my project window. Then at the Inspector tab, in the "Rig" settings of Character Import Settings, set the "Animation Type" to "Humanoid" from "generic".

And click “Apply” and the configure button. In the Inspector tab, a human shape with green dots appears:



Green means the bones are correctly connected. Next, we hit “done” button and go to the Animation settings, select the loop time box for “Idle”, “Walk” and “Run”.

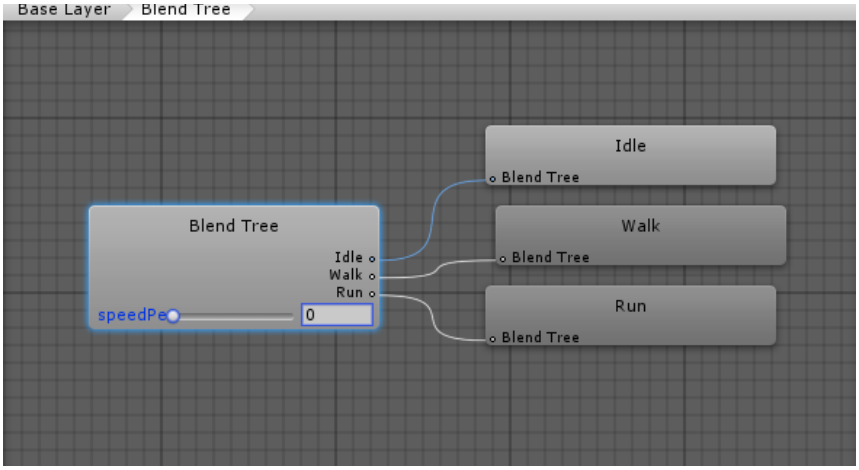
Then there appears a prefab named “Character” in the Asset folder, as the following picture shows:

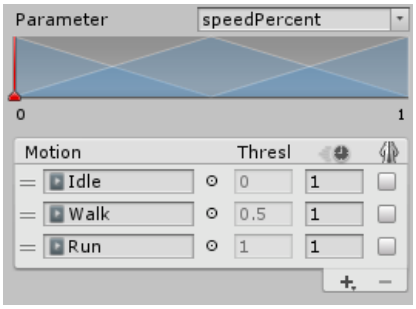


We just drag the prefab into the scene to finish importing our robot model.

9. Make the robot move

We need to create an animator controller in the Asset folder tab first, and associate it with the character we just import. Then, inside the animator we build up a blend tree with a variable speedPercent. This variable will be created later in the PlayerController script, to let the character do the correct motion under corresponding speed. The following pictures show the settings of the blend tree:





Next, we create and attach a C# script to our character. We name the script “PlayerController.cs”. We mainly work on this script to make our character walk, run and jump.

The following code shows how we active the animator to make the robot model animated. When holding the shift key, the character will begin to run:

```
//hold shift key to run
bool running = Input.GetKey(KeyCode.LeftShift);
```

```
//animator
//if running is true, take the running speed, or take the walking speed
float animationSpeedPercent = ((running) ? currentSpeed/runSpeed : currentSpeed/walkSpeed * 0.5f);
animator.SetFloat("speedPercent", animationSpeedPercent, speedSmoothTime, Time.deltaTime);
```

The above code is put in the void Update() method.

The following code shows how to control the character to walk and run:

```
void Move(Vector2 inputDir, bool running){
    if(inputDir != Vector2.zero){
        float targetRotation =  Mathf.Atan2(inputDir.x, inputDir.y) * Mathf.Rad2Deg;
        transform.eulerAngles = Vector3.up * Mathf.SmoothDampAngle(transform.eulerAngles.y, targetRotation, ref turnSmoothVelocity, turnSmoothTime)
        //print(inputDir.x+";"+inputDir.y); //for testing
    }

    //when we don't have an input, target speed will be zero
    float targetSpeed = ((running) ? runSpeed : walkSpeed) * inputDir.magnitude;
    //print("magnitude:"+inputDir.magnitude);

    //for jumping
    velocityY += Time.deltaTime * gravity;
    //for smoothing the speed
    currentSpeed = Mathf.SmoothDamp(currentSpeed, targetSpeed, ref speedSmoothVelocity, speedSmoothTime);

    //in three directions
    Vector3 velocity = transform.forward * currentSpeed + Vector3.up * velocityY;
    //move the character
    controller.Move(velocity * Time.deltaTime);

    //speed in x and z directions
    currentSpeed = new Vector2(controller.velocity.x, controller.velocity.z).magnitude;

    //don't do jump
    if(controller.isGrounded){
    velocityY = 0;
    }

}
```

```
//call move() with the inputDir
Move(inputDir, running);
```

We create a method move(), and call it in the void Update(). The variable inputDir is the distance to go in x and z directions. To make the character walk or run, firstly we should turn the character into the correct direction. transform.eulerAngles is set to do this. The variable “gravity” and “velocityY” are for jumping. We’ll define the jump() method very soon. The value of inputDir can come from the arrow keys of the keyboard, so right now, we can control the robot model to walk and run using the arrow keys and shift key. The character will turn to the right direction and then do the walking or running motion. To make the turning and the changing of speed appear smoothly and naturally, we define some variables and use the SmoothDamp method.

The following code is how we define the jump method and call it in the void Update() method:

```
//jump
void Jump(){
    if(controller.isGrounded){
        float jumpVelocity = Mathf.Sqrt(-2.0f * gravity * jumpHeight);
        velocityY = jumpVelocity;
    }
}
```

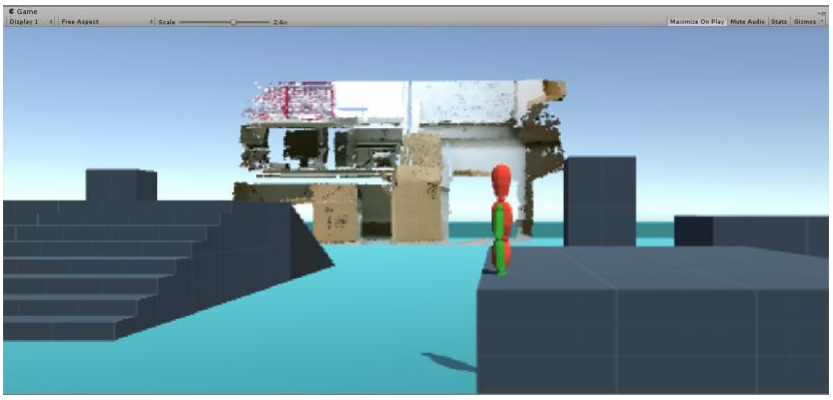
```
//space key to control the jump
if(Input.GetKeyDown(KeyCode.Space)){
    Jump();
}
```

The space key is set to trigger the jumping. We apply physical laws to get the velocity in Y direction.

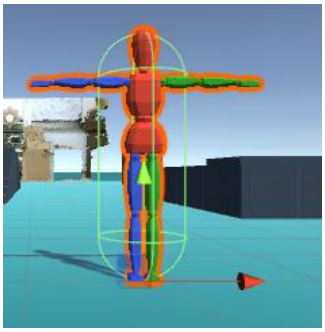
Now, we can control our character to walk, run or jump using keyboard.

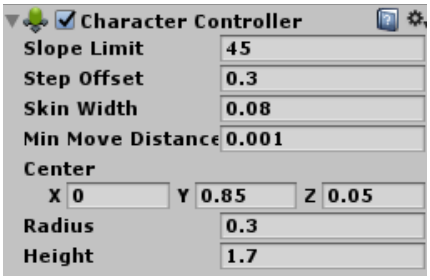
10. Set Colliders

We have done a lot to make the robot model animated and move. But without setting the collider to the character and the environment, when the robot model runs into a solid object, it will really run into the object, which doesn't obey the physical laws, as the following picture shows:



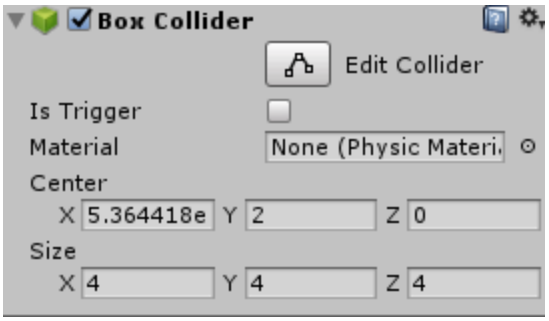
To solve this problem, we firstly add a Character Controller component to the character and do the following settings:



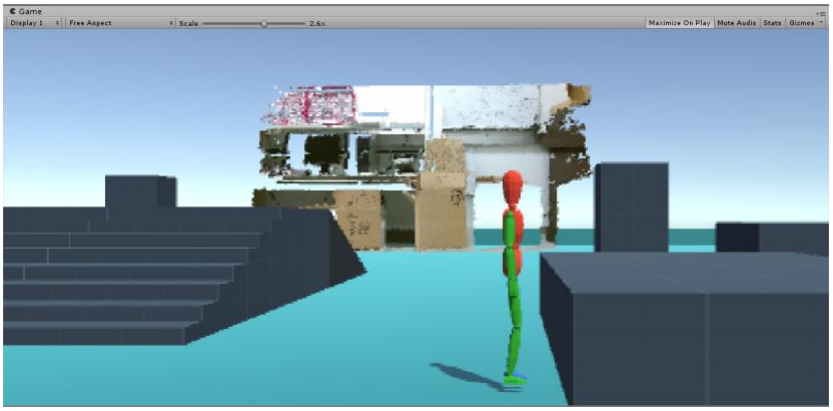


It will form a collider in the shape of the green line as the above picture shows.

Then we attach a box collider to every solid object in our scene. We double check and make sure the floor and walls also possess this kind of collider:



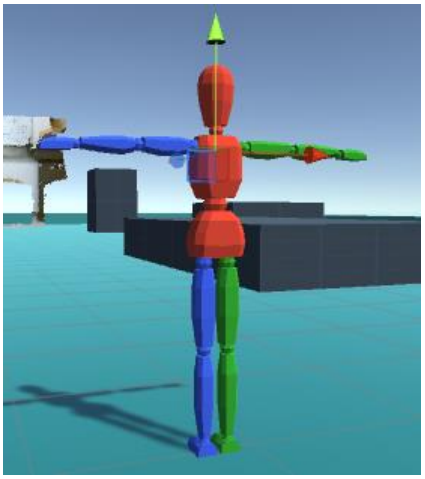
After setting the colliders, when running into a solid object, the character can't penetrate and will stop.



There exists a problem with the point cloud. The mesh collider for the point cloud seems not working at all. Maybe that's because we have too many points (more than 1 million) and the hardware and software can't handle it. So, after several attempts, I just manually set a few box colliders into the point cloud scene, according the actual shape. The box colliders I set manually are not very precise, but it's better than no collider at all there. The following picture shows that when the character tries to walk into the virtual box, it just hit the collider I set manually and stops:

11. Set a third-person view Camera

We create an empty child object under “Character” and name it “target look”. We make the position situate at the chest of the robot model:



That’s where the camera will look and follow. Then, under “Main Camera”, in the Inspector tab, we add a script named “ThirdPersonCamera”. The code is very simple, we define a variable to hold the Target look, get and update its position according to the target position. We also have a variable to control the distance from the camera and the target. We set it to 7 for the initial value:

```
public Transform target;
public float dstFromTarget = 7.0f;
```

To view the whole script, please refer to the Resource folder and check the “ThirdPersonCamera.cs” file.

After setting this camera view, when we play the scene, our view will follow the robot model as a third-person view.

12. Click and move

When clicking on screen, the mouse position we get is a 2D position. To convert it into a 3D position in the scene as a target position to go to, we construct a “Ray” and perform a “Raycast”. Basically, we cast a ray into the scene, from the 2D screen location we click at. If the ray collides with something solid in the scene, the collision position will be stored and it becomes our target position to go if the ray hits the plane.

This is the reference document: https://docs.unity3d.com/560/Documentation/Manual/nav-MoveToClickPoint.html .

And the main code is as the following (with an indicator to mark the position to go, which we will implement in the next step):

```
//click to move
if(Input.GetMouseButtonDown(0)){
    //show the indicator
    indicator.GetComponent<Renderer>().enabled = true;
    //print("MouseDown"); //for testing
    //calculate a 3D point according to the mouse click position
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hitInfo = new RaycastHit();
   //hitInfo contains the 3D position
    if (Physics.Raycast(ray, out hitInfo)){
        if (hitInfo.collider.tag == "plane"){
            indicator.position = new Vector3(hitInfo.point.x, hitInfo.point.y + 0.3f, hitInfo.point.z);
            target = hitInfo.point;
            isOver = false;
            //print(target); //for testing
        }
    }
}
```

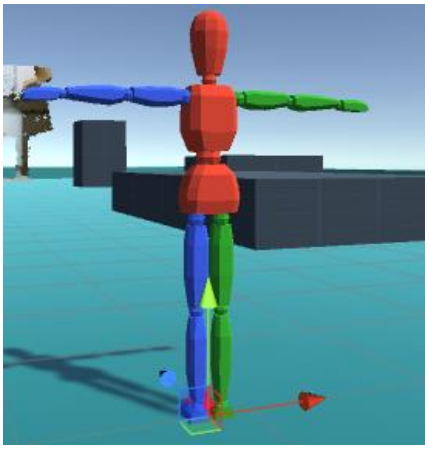
The code is put in the void Update() method in the PlayerController.cs script. To make it work, we should tag the floor and other planes the character can move to, like stairs and ramp, as “plane”.

13. Create an indicator

We want an indicator to mark the target position to go. For doing this, we create 2 public variables in the PlayerController.cs firstly:

```
public Transform player;
public Transform indicator;
```

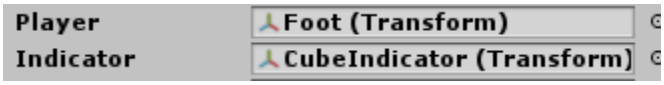
Player is to mark the position of the robot model and indicator is for the position of the indicator. Then we create an empty child object under “Character”, name it “Foot”, and make it at the very bottom of our robot model to indicate the position of the foot:



We drag it into the box of variable “player” under “Character” in the Inspector tab in the Player Controller Script section.

Then we create, in the scene, a cube object as the indicator. We make it an appropriate size and color, and name it “CubeIndicator”. Then as what we do for the “player” variable, we associate this cube object with the variable “indicator” in the PlayerController.cs.

We have this in the Inspector tab under “Character” now:



And we go to the PlayerController script to add the following code:

```
//if the character hasn't reach the target
if(!isOver){
    Vector2 offset = new Vector2((target.x - player.position.x),(target.z - player.position.z));
    //inputDir is the distance to go, in x and z directions
    inputDir = offset.normalized;

    //if the character reaches the target
    if(Vector3.Distance(target, player.position) < 0.3f){
        isOver = true;
        //remove the indicator
        indicator.GetComponent<Renderer>().enabled = false;
        //clear the inputDir
        inputDir = Vector2.zero;
    }
}
else{
    //update the inputDir
    Vector2 input = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
    inputDir = input.normalized;

}
```

“isOver” is a Boolean variable indicating if the robot model reaches the target position. If it doesn’t, the indicator will still be rendered, until the target is reached. “inputDir” is the distance to go, it’s a 2D vector in x and z directions.

14. Add sound effect

We just want to make a basic sound effect for the robot model. We implement different sounds when the robot walks, runs and jumps.

Firstly, we go to the asset store to import some step sounds. There are quite a lot of nice free ones there. I use this asset: https://www.assetstore.unity3d.com/en/#!/content/2924 . We can import the sound files from another site, too.

In the Hierarchy tab, under Scene --> Character, we build three empty objects: AudioWalk, AudioRun and AudioJump. Right in the Inspector tab, we add component for each of the object: Audio -> Audio Source, and drag a suitable sound file from the asset folder tab to the “AudioClip” area in the Inspector tab. Then we go back to the character and define three audio sources in PlayerController.cs script as the following:

```
public AudioSource audioWalk;
public AudioSource audioRun;
public AudioSource audioJump;
```

Next, in the Inspector tab for the Character, there appear three sound source variables to be defined: Audio Walk, Audio Run and Audio Jump. We drag the corresponding object that contains the audio source files into the text area, like the following:

At last, we add some scripts into the PlayerController.cs file. We create a function PlayFootSteps(), and call the function in void Update(), at the end, after move(), jump() and the animator are called. In the PlayFootStep() function, we basically switch on or off the sound and determine how the sound loops, under different moving speed. For example, the following is how we play the sound when the character is running, only the audio for running are enabled and set to loop:

```
if(currentSpeed > walkSpeed + 0.1f && currentSpeed < runSpeed + 0.1f){
    audioWalk.enabled = false;
    audioWalk.loop = false;
    audioRun.enabled = true;
    audioRun.loop = true;
    audioJump.enabled = false;
    audioJump.loop = false;
}
```

15. Observations and Conclusions

The following picture shows that the robot model is walking into the 3D point cloud scene by chasing the red target indicator:

The result is exported as an .exe file in the Result.zip file. Clicking the Multipj.exe file will run the result of the project. We can use arrow keys to move the robot model, or click on the floor to set a target position. To make the robot run, we hold the shift key. To let the robot jump, we press the space key.

The result of the project meets the goal. I successfully reconstruct a 3D point cloud scene from the given RGB-D dataset and import it into Unity. I bring a 3D robot model into the scene, and we are able to animate the robot, to make it walk or run to a marker position by clicking in the scene. The robot model can be controlled by using keyboard, too, and it moves with a sound effect.

This project involves many different tools and concepts at once. Unlike the project of other courses, during which the tools or language we use will usually be consistent, this project involves C++ and C#, and the following tools and libraries: OpenCV, Point Cloud Library, Ubuntu System, CMake, MeshLab, Unity5, blender (if we want to make our own robot model). We should have the ability to understand all the concepts and integrate those tools together to get a satisfied result of the project.

One of the hardest part is to build the environment to convert the 2D file to 3D, using OpenCV and PCL. At first, I try to do it directly on Windows, but there might be a version problem between OpenCV and PCL themselves. I can successfully run OpenCV or PCL, individually, but it seems I have to choose appropriate versions of them to let them work together. I always get some dependency issues when I try to use both OpenCV and PCL. Then my classmate suggests me to build an Ubuntu system, and then install the tools on Ubuntu. I'm very happy that I follow her advice. The installment and implementations of these libraries on Ubuntu are way much easier to do. This is the first time I try a Linux based operating system. And now I understand why so many people are fond of it. I'd like to get down and learn more about Linux in the future, too.

Another issue I encountered is in Section 5, when we choose a format of the point cloud file. At first, I use ::io::savePCDFile() in the tutorial to save a .PCD file. But the Point Cloud Viewer in Unity only reads .OFF file. Then I find the software MeshLab, which can export a .OFF file. However, it doesn't read .PCD file. I check the format list that MeshLab can read, luckily there exists .PLY format. Then it takes me some time to implement a C++ script to convert the PCD file to PLY file. After all of these, I ask myself, PCL library is powerful, isn't there any possibilities that PCL supports PLY file? The answer is simply yes. According to the official document, we can use ::io::savePLYFileBinary() to directly get a PLY file. That saves me a big step in the project. But I really take some time to realize and find it.

In a word, the project is comprehensive and after doing it, I learned a lot. Not only how to use the tools, but also how to choose an appropriate tool and how to integrate multiple tools, to solve problems efficiently.

16. References and useful Tutorials for this project

In Section 3, for understand Pinhole camera model and intrinsic matrix:

http://ksimek.github.io/2013/08/13/intrinsic/

In Section 4, very useful Tutorial to convert from 2D image to 3D point cloud file (part II of this tutorial):

https://github.com/gaoxiang12/rgbd-slam-tutorial-gx

In Section 6, Point Cloud Free Viewer to import the point cloud into Unity:

https://www.assetstore.unity3d.com/en/#!/content/19811

In Section 7, basic official Unity tutorial "roll a ball":

https://unity3d.com/learn/tutorials/s/roll-ball-tutorial

In Section 8 and 9, video tutorial for creating character using Blender and controlling the character in Unity:

https://www.youtube.com/playlist?list=PLFt_AvWsXl0djuNM22htmz3BUtHHtOh7v

In Section 12, using ray and raycast to implement the move to click functionality:

https://docs.unity3d.com/560/Documentation/Manual/nav-MoveToClickPoint.html

In Section 14, footsteps sound free asset:

https://www.assetstore.unity3d.com/en/#!/content/2924

17. Contents in the Resources folder

Section01.original dataset and intrinsic matrix:

- color.jpg
- depth.png
- IntrinsicRGB
- Read me.txt

Section04.cpp code to convert 2D to 3D and the result file:

- convert.cpp
- plyfile.ply
- plyfileBeforeOptimization.ply
- pointcloudfile.off

Section08.robot model file used in the project

- Character.blend
- Read me.txt

Section09. PlayerController script

- PlayerController.cs

Section11. third-person view script attached to main camera

- ThirdPersonCamera.cs