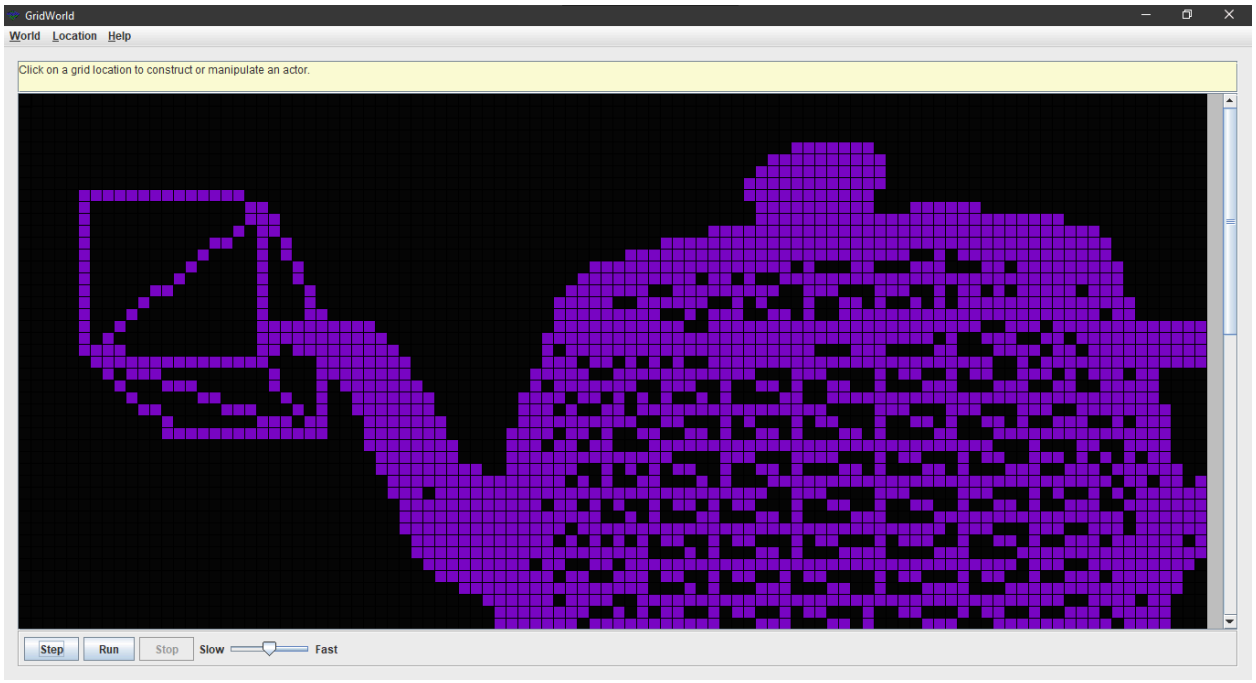
Cy Ranch CS2: GridWorld 3D Engine

Eric Alfaro

Lab Objectives

In this lab, you will create a 3D engine using the AP GridWorld project. You will be expected to create a scene interface that can draw lines and triangles with pixels as actors. You will use Object Oriented Programming to define classes such as Vector and Triangle to represent 3D structures. You will also be expected to implement a simple importer of .obj files and a rasterizer to draw 3D shapes on a 2D grid.



Example Execution

Background Knowledge

3D engines are programs that can convert data about 3D objects and render them on a 2D screen. There are many techniques, such as ray tracing and ray marching. You will implement a rasterizer, which is the most common 3D rendering technique. Rasterizers represent data as groups of vertices with an x, y, and z coordinate. These typically connect to form triangles, as they are the easiest primitive shape to perform mathematical operations to calculate shading and other properties. The rasterizer converts this data into a 2D image which is the result observed in many movies and video games.

Because 3D points cannot be represented on a 2D screen, the z coordinate is instead used to manipulate the x and y coordinates to create an illusion of depth. This is typically done through a projection matrix, which will be explained later in the lab.

It should be noted that there are many more sophisticated forms of rendering and drawing that will not be explored in this lab. This will only serve as an introduction to the math and underlying knowledge about 3D rendering.

Outline

<table>
<tr><td>Part A</td><td>Define classes to draw on a 2D grid

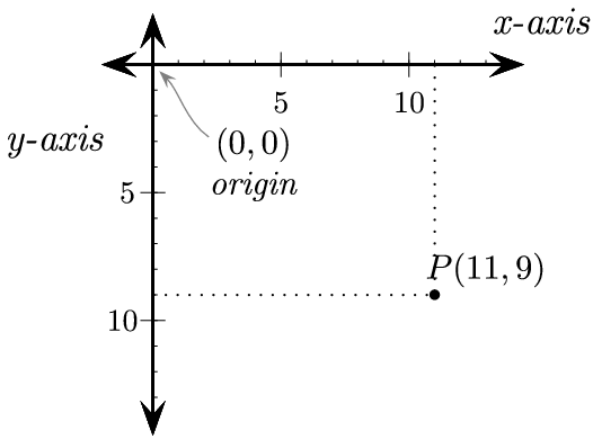
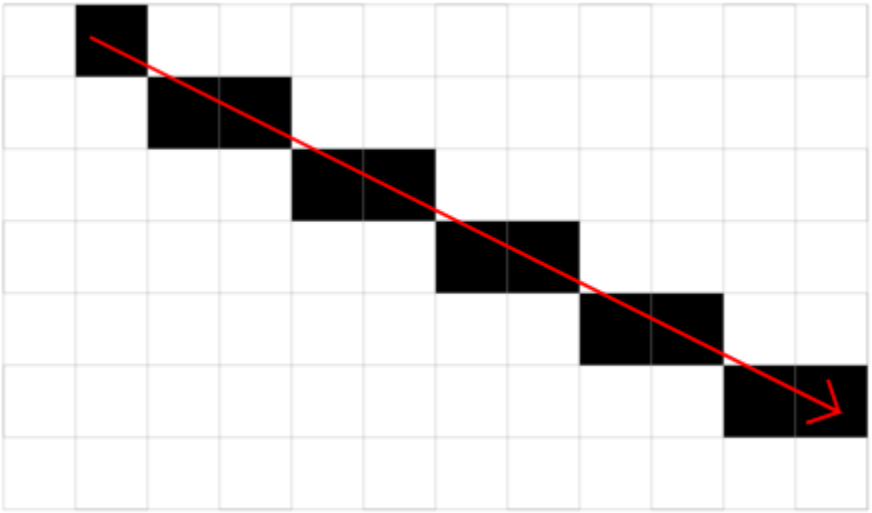
Classes:
* <code>PixelActor extends Actor</code>
* <code>public PixelActor()</code>
* <code>ScreenWorld extends ActorWorld</code>
* <code>public ScreenWorld(int width, int height)</code>
* <code>@Override public void step()</code>
* <code>public void drawLine(double x1, double y1, double x2, double y2)</code>
* <code>public void drawTriangle(double x1, double y1, double x2, double y2, double x3, double y3)</code>
* <code>public void setPixelColor(int x, int y, Color c)</code>
* <code>public void clear()</code>
* <code>public int getWidth()</code>
* <code>public int getHeight()</code>
* <code>public List<Mesh> getScene()</code>
* <code>MainRunner</code>
* <code>public static void main(String[] args)</code></td></tr>
<tr><td>Part B</td><td>Define classes to represent a 3D space

Classes:
* <code>Vector</code>
* <code>public Vector()</code>
* <code>public Vector(double x, double y, double z)</code></td></tr>
</table>

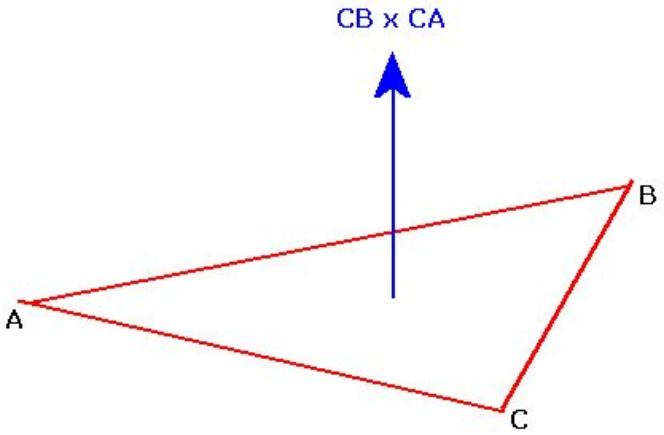
	* `public String toString()` * `public Vector subtract(Vector other) {` * **Triangle** * `public Triangle(List<Vector> points)` * `public List<Vector> getPts()` * `public void setPts(List<Vector> p)` * `public Vector getPts()` * `public void calculateNormal(List<Vector> transPts)` * **Mesh** * `public String toString()` * `public Vector getRotation()` * `public void setRotation(Vector r)` * `public Vector getTranslation()` * `public void setTranslation(Vector t)` * **final ObjImporter** * `public static Mesh importObj(Scanner s)`
Part C	Define methods to rasterize 3D objects to the ScreenWorld class Classes: * **Camera** * `public static void rasterizeMesh(ScreenWorld s, Mesh m)` * `private static Vector rotatePoint(Vector p, Vector rotation)` * `private static Vector translatePoint(Vector p, Vector translation)` * `private static Vector rasterizePoint(Vector p, double screenSize, double angle)`

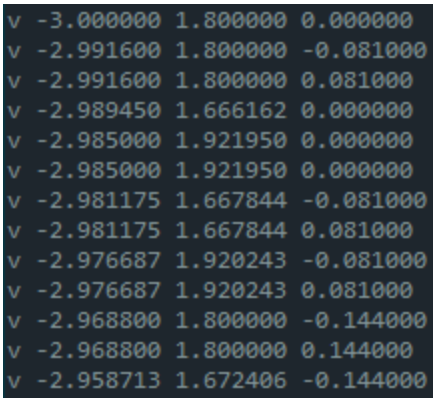
Instructions

Step	Description
A0	**Installing GridWorld API** GridWorld is a free jar file of classes created by College Board that can be used to quickly set up a grid project with a user interface. This link has instructions on how to install the classes into your IDE of choice. Familiarize yourself with the Actor and World classes before continuing.
A1	**Creating the ScreenWorld and PixelActor classes** Start by creating the PixelActor and ScreenWorld classes as described by the outline.  PixelActor examples The PixelActor will only have a constructor that resets the Color (from java.awt) to black. ScreenWorld will have the private instance variables width and height. Use these and the parameterized constructor to set this world's grid to a new BoundedGrid (from info.gridworld.grid). Initialize each cell in the BoundedGrid with a new PixelActor. ScreenWorld should have a list of meshes for a scene, which contains all the meshes that will be rasterized in one step later on. Initialize this in your main constructor. Override the step method from ActorWorld. This will be used to rasterize the scene every frame later on.

A2	**Creating the draw methods** ScreenWorld should have a method setPixelColor, which updates the color of the PixelActor at coordinate (x, y) to an argument color.  Coordinate System used in GridWorld ScreenWorld should have the drawLine method, which takes a start coordinate (x1, y1) and end coordinate (x2, y2) and activates the pixels that represent the actual line. This is most effectively done using a “Digital Differential Analyzer” algorithm, as described here. You may not assume that x1 will be less than x2 or that y2 will be more than y1; write your code to work for lines in any direction. 

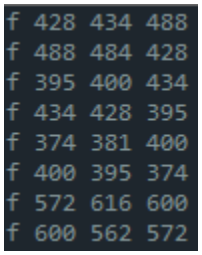
	drawLine example (from cs.virgina.edu) Hint: Use Math.floor() to get the pixel that a point resides in. ScreenWorld should have a drawTriangle method, which takes three coordinates [(x1, y1) (x2, y2) (x3, y3)] and uses the drawLine method three times to complete a triangle. ScreenWorld should have a clear method, which resets all pixelActors to black.
A3	**Creating the MainRunner class** This class will be the entry point of your program. You do not need to implement other methods here. Use the main method to create a ScreenWorld and test your functions from part A1 and A2.
B0	**Creating the Vector class** The Vector class represents a point in 3D space. It should have an x, y, and z coordinate, which are all public and double. While it is usually good practice to utilize encapsulation for client classes, leaving the coordinates as public is convenient for writing calculations in an easy to read manner. Create a default constructor that sets all values to 0 and a parameterized constructor. The Vector class should have a subtract method, which returns a new Vector that has been subtracted by the other parameter.
B1	**Creating the Triangle class** The triangle class is a list of three points. It will be used to build meshes later on. Because triangles have only three points, all points of a triangle will always be coplanar. This allows us to define a single vector to represent the direction of the triangle face. This property will be used to exclude backwards facing triangles during rendering.

	 Triangle normal (from stackoverflow) The normal of a triangle is defined as the cross product of two of their sides. This is most easily done using Newell's method as described here. Implement this method within the triangle's calculateNormal method, which should take a list of transformed points (rotation and translation, which will not be implemented at this stage) to update its own normal property.
B3	**Creating the Mesh class** A mesh is a collection of triangles to represent a 3D model. It should contain two vectors to represent translation (the position of the model) and rotation. Both properties will be applied to each triangle in the next part of the lab.
B4	**Creating the ObjImporter class** This program will take 3D data as .obj files, which is one of the most popular storage formats due to its simplicity. The file first defines a list of vertices with an x, y, and z coordinate as shown below.



A peek into a .obj file

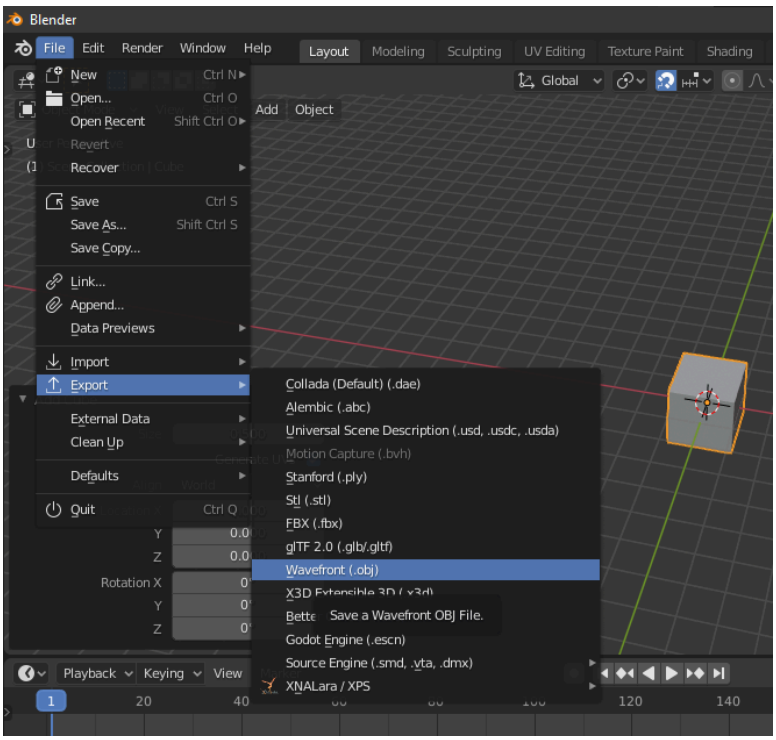
The position of the vertex within the list is its index. The file then defines a list of faces, which are defined by integer indices (1-indexed) of the vertex list defined above.



A list of faces in a .obj file

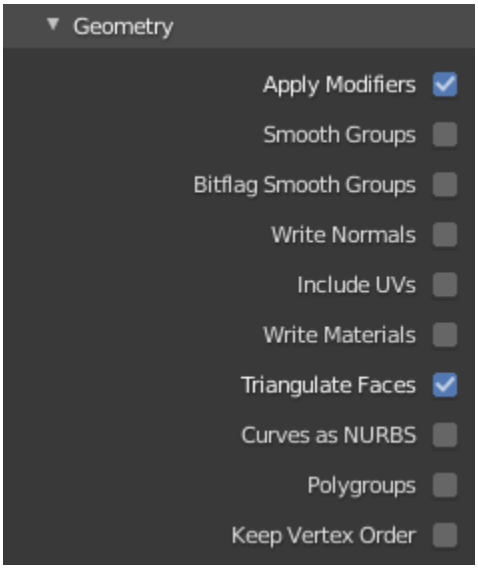
Use these patterns to create a mesh object from a file. You do not need to do anything with lines in a .obj file that do not start with a ‘v’ or an ‘f’.

If you want to import meshes into your program, you can use Blender’s obj exporter.



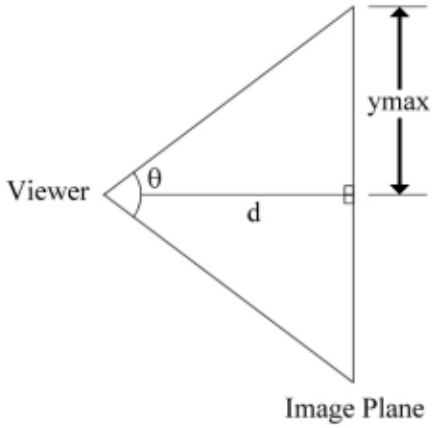
Exporting a model

Ensure that you select the matching geometry flags when exporting to prevent errors in your program.



Export flags

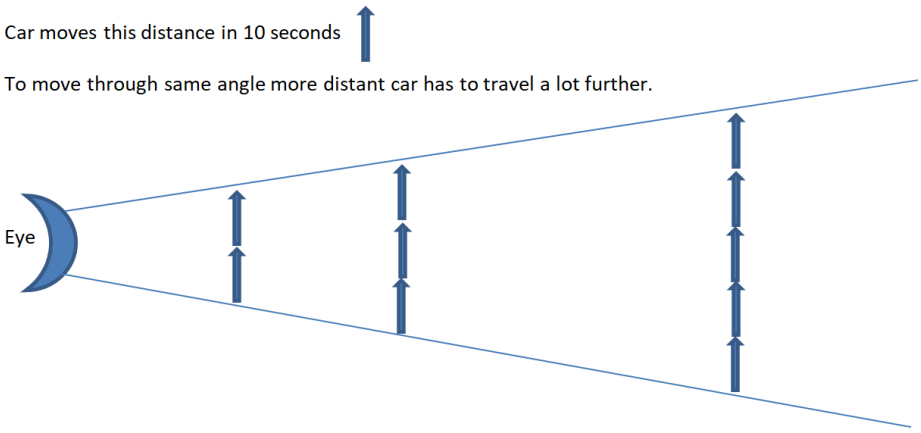
C0	**Creating the Camera class** The camera will be where the actual rasterization of the mesh occurs. Create the translatePoint method, which returns a new point that has been translated by the translation vector argument. In your camera class, define a rasterizeMesh method that will call the methods detailed in later parts of C in order to draw triangles onto a ScreenWorld.
C1	**Projection** To transform a 3D point to be represented onto a 2D screen, we must use the x, y, and z coordinates to represent depth. This is most easily done using a projection matrix, which is multiplied by each point vector. As you can see below, this is a 4x4 matrix, while our points are a 1x3 matrix. We need to define a fourth value, w, which can be used for different transformations on our 3D coordinates. This value is always 1 for 3D points. $$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$ A common projection matrix. Aspect ratio (or aspect) is a value used to scale the projected point to the correct size of the screen. This value may be excluded, as our screen will always be 100x100, or an aspect ratio of 1. Field of view (or fov) denotes the width of the camera's view of the world. We will define this as a constant in our camera class with the value of $\frac{2\pi}{3}$ radians. We take the inverse tangent of this value in order to increase or decrease the scale of the viewing plane depending on the size of our field of view.



Visual representation of field of view

Far and near are values to clip points with certain z distances that are either too close or too far to the camera. This may be excluded in our program.

Note that in the fourth column of the projection matrix, we multiply a constant value by the z coordinate of the point. This is used to extract the z value from a projected point, as the z coordinate of a point is the most crucial aspect to simulating depth.
If you are ever traveling in a car or train, you may notice that objects that are farther away appear to move at a slower rate relative to the objects closer to you. We can express this relationship by dividing our projected x and y positions by -z, as larger z values will lead to smaller changes in x or y, while smaller values will lead to larger changes in x or y.



Intuition behind dividing our x and y by z

Implement the above matrix in your rasterizePoint method. You do not need to define matrix structures or multiplication methods; you may simply take the x, y, and z values and multiply them by the appropriate constants.

C2

Rotation

Rotation of 3D points follows a similar process to projection. We will multiply each point by a rotation matrix for each direction of our rotation vector.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

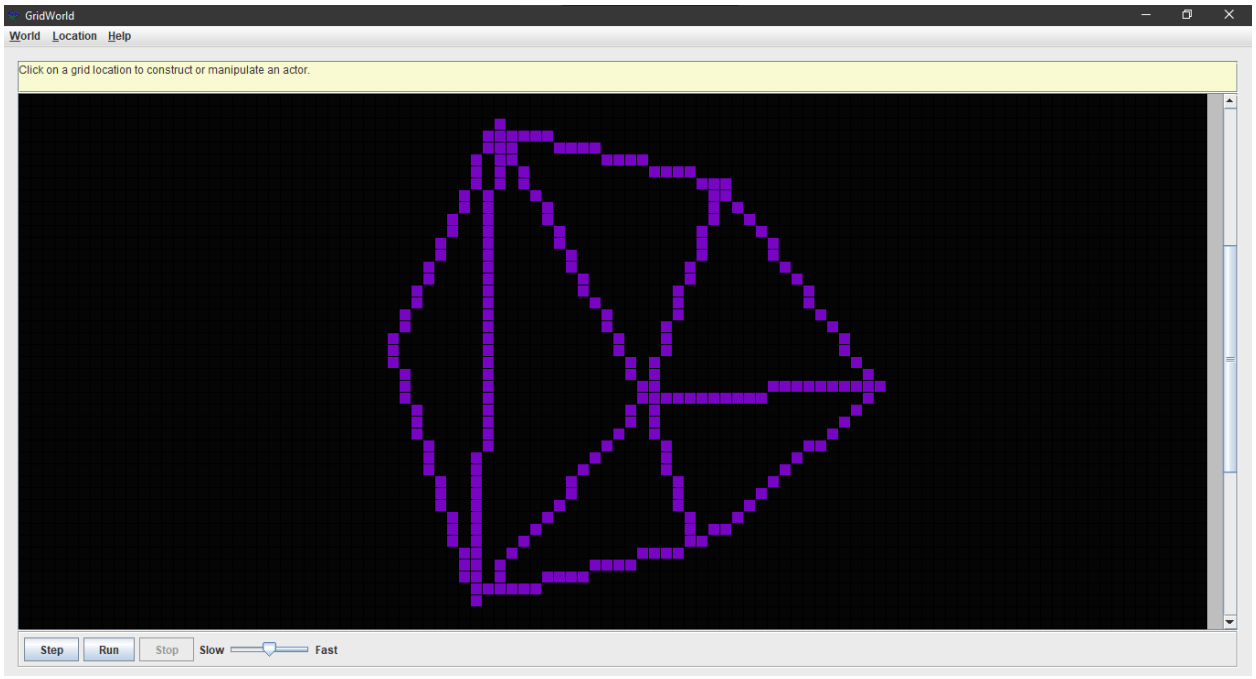
$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrices for x, y, and z rotation

C3	**Combination of lab components** At this point in the lab, you should have all the components necessary to rasterize a 3D model. Complete the camera's rasterizeMesh method by taking the point of each triangle and applying rotation, translation, and projection methods on each point. Use the normal of the triangle to decide if the face should be drawn or not (hint: if the normal value's z component is negative). Draw each triangle using ScreenWorld's drawTriangle method. Use the MainRunner class to import .obj files and add them to a ScreenWorld scene. Implement ScreenWorld's step method to rasterize each mesh of the scene using the camera's rasterizeMesh method. Test your rotation method by updating the rotation of each mesh within your scene in the step method.

Final Result



Full solution available at https://github.com/kiwijuice56/ap-gridworld-3d-engine