Project3F17

From Immersive Visualization Lab Wiki

Contents

Project 3: Robot Army

In this project you will need to implement a scene graph to render an army of Android-inspired robots.

The total score for this project is 100 points. Additionally, you can obtain up to 10 points of extra credit.

1. Sky Box (25 Points)

Start with code that uses your trackball code, and modify it to control the camera instead. (If you didn't get that to work, keyboard controls will suffice.)

Create a sky box for your scene with the robots. A sky box is a large, square box which is drawn around your entire scene. The inside walls of the box have pictures of a sky and a horizon. Sky boxes are typically cubic, which means that they consist of six square textures for the six sides of a cube. Here (http://learnopengl.com/#!Advanced-OpenGL/Cubemaps) is a great tutorial for sky boxes in modern OpenGL.

Here is is a nice collection of textures for sky boxes (http://www.f-lohmueller.de/pov_tut/skyboxer/skyboxer_3.htm) , and here is an even bigger one (http://www.custommapmakers.org/skyboxes.php) .

Draw a cubic sky box and make it extremely big. For instance, by giving it coordinates like -1000 and +1000.

Make sure single-sided rendering (triangle culling) is enabled with these lines somewhere in your code to ensure that you will never see the outside of the box (this assumes that your sky box is defined with the triangles facing inward):

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Use the following settings for your texture after your first `glBindTexture(GL_TEXTURE_CUBE_MAP, id)` for correct lighting and filtering settings:

```
// Make sure no bytes are padded:
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // Deprecated in modern OpenGL - do not use!

// Select GL_MODULATE to mix texture with polygon color for shading:
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE); // Deprecated in modern OpenGL - do not us

// Use bilinear interpolation:
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Use clamp to edge to hide skybox edges:
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

To familiarize yourself with texture mapping in OpenGL, we provide sample code, which loads a PPM file and uses it as a texture for a quad. If you decide to use one of the above referenced sky box images, you will have to convert them from JPEG to PPM format. The free image processing tool IrfanView (http://www.irfanview.com) for Windows will do this for you. Alternatively, you can use a third party library such as SOIL (http://lonesock.net/soil.html) to natively load JPEG images.

Grading:

- 5 points for functional camera controls with keyboard or mouse
- 5 points for the sky box without textures
- 5 points for the textures
- 5 points for correct rendering of edges and corners (seamless edges)
- 5 points for correct culling of the skybox

2. Scene Graph Engine (20 Points)

To connect the parts of the robot (head, torso, limbs, eyes, antennae), we need to first implement a simple scene graph structure for our rendering engine. This scene graph should consist of at least three nodes: `Node`, `Transform` and `Geometry`. You are free to add more scene graph node types as you see fit.

- Class `Node` should be abstract and serve as the common base class. It should implement the following class methods:
  - an abstract draw method: `virtual void draw(Matrix4 C)=0`
  - an abstract `virtual void update()=0` method to separate bounding sphere updates from rendering (4 points)
- `Transform` should be derived from `Node` and have the following features: (8 points)
  - store a 4x4 transformation matrix M
  - store a list of pointers to child nodes (`std::list<Node*>`)
  - provide class methods to add and remove child nodes (`addChild()`, `removeChild()`) from the list
  - its draw method needs to traverse the list of children and call each child node's draw function
  - when `draw(C)` is called, multiply matrix M with matrix C.
- `Geometry` should be derived from `Node` and have the following features: (8 points)
  - set the modelview matrix to the current C matrix
  - an initialization method to load a 3D model (OBJ file) whose filename is passed to it (`init(string filename)`. Your OBJ loader from project 2 should work.
  - have a class method which draws the 3D model associated with this node.

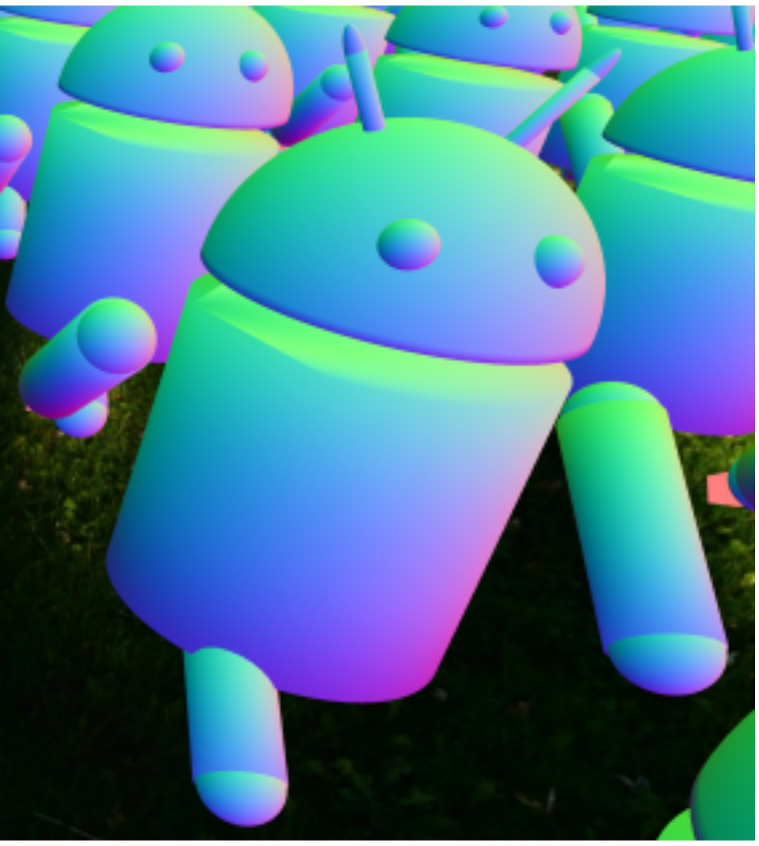
3. Walking Android Robot (25 Points)

Now that we have the scene graph classes, it is time to put them to work, and build a robot with them.

Thanks to our tutor Yining Liang, who created these parts for his own CSE 167 homework project, you have the following robot parts to choose from: head, body, limb, eye, antenna. You will find the OBJ files in this ZIP file.

Build your own robot using the `addChild` methods. Use at least 3 different types of parts for your robot (e.g., body, head and limb). In total, your robot needs to consist of at least 4 parts, 3 of which need to be moving independently from one another and they need to be connected to the 4th part. (15 points)

This is an example of a valid robot with two antennas, two eyeballs, one head, one torso and 4 limbs (2 legs and 2 arms):



Use your creativity to build the most creative robot in class! The 5 most creative robots in class, after a vote on Piazza, are going to get extra credit.

Once you've created your scene graph, you need to get your rendering engine ready to recursively traverse the scene graph for rendering by creating a root node of type Group and calling its draw() function with the identity matrix as its parameter.

Animate the robot to make it look like it is walking, by changing the matrices in the Transform nodes. (10 points)

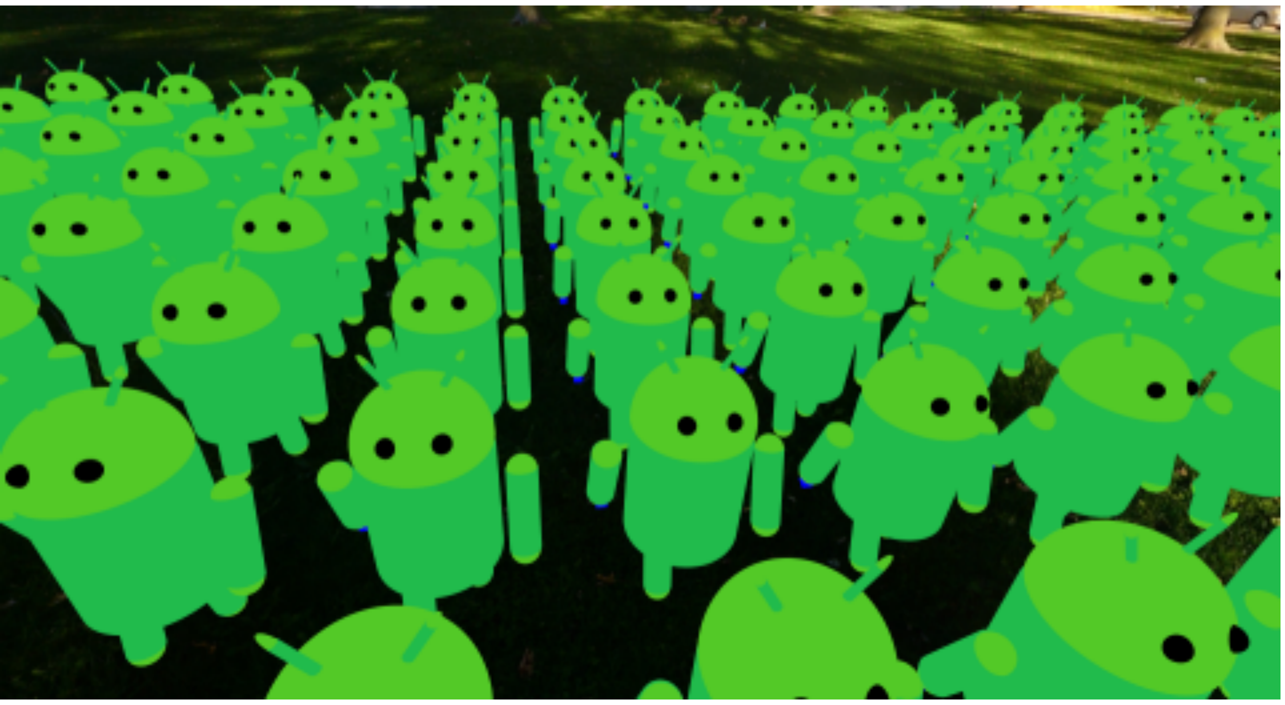
4. Robot Army (15 Points)

Test your implementation by constructing a scene which consists of a large amount of robots, at least 100. The robots can all be identical clones.

- Distribute the robots on a 2D grid (i.e., place them on a plane with uniform spacing). For 100 robots, use a 10x10 grid. (10 points)
- Enable the animation for all the robots so that they look like they are walking. (3 points)

- Enable your rotation and scale routines (keyboard or mouse) to allow rotating the grid of 3D objects and zoom in or out. (2 points)

This image illustrates the grid layout of the robots:



5. Culling (15 Points)

Implement object level culling, to allow the existence of thousands of instances of your robot, without having to render them all at once.

Determine the parameters for a bounding sphere (Vector3 for its center point, and a radius) for each of your robots, which contains all parts of the robot. Add an option to toggle the rendering of the bounding spheres on or off with a keyboard key. You can render the spheres by using your point rendering algorithm from project 1 and the sphere OBJ from project 2. Other ways of rendering spheres are also acceptable as long as you can see the robot within the sphere. (5 points)

Note: You do not need to find the tightest possible bounding spheres - that can be rather difficult. Just make them as tight as you reasonably can.

Add view frustum culling using the bounding spheres of the objects. If the bounding sphere of an object is completely outside of the view frustum, the object should be culled (not rendered). Your culling algorithm should make use of a utility function to test whether a bounding sphere intersects with a given plane (the planes of the view frustum), or whether the sphere is entirely on one side of the plane. Enable a keyboard key to turn culling on and off. (8 points)

Increase the amount of robots by orders of magnitude, by creating a larger array of them. A good portion of the array should be off screen for the culling to be effective. Display the rendering time per frame in your text window, and show that by turning culling on your rendering time decreases. (2 points)

6. Extra Credit (Max. 10 Points)

a) Create an editor for robots. It must include functionality to interactively: select a limb, place it, set its initial angle, set an angular range for its motion, and set the velocity of its swing. (5 points)

b) It's easier to calculate tight axis aligned bounding boxes than tight spheres, but culling is harder with bounding boxes. implement view frustum culling using tight bounding boxes (in world coordinates) in place of spheres. You also need to create a demo mode in which you zoom the camera out (increase the FOV) but do the culling with the original FOV, so that one can see when the robots get culled. (5 points)

c) This can be done in conjunction with b) using tight bounding boxes, or with the original bounding spheres: create a hierarchical culling algorithm by storing bounding box/sphere information at every level of the scene graph, so that you can cull an entire branch of the scene graph at once. Structure your scene graph so that you have multiple levels (for instance, by subdividing your army into four quarters, and create a node above each quarter army. Also use the increased FOV like in part b) to demonstrate what gets culled when. (5 points)

7. Creativity Contest (Up to 5 Points)

We're going to have a contest for the top 5 most creative robot creations. Submit a JPEG image or GIF animation of your robot to Piazza by the deadline. Instructions will be posted on Piazza. To create a GIF animation you can use this converter (http://gifmaker.me/) . To convert any image file format to a GIF image, we recommend IrfanView. (http://www.irfanview.com/) The top five most voted for robots are going to get extra credit.

The winner of the contest will get 5 points of extra credit. The second will get 4 points, third gets 3, fourth 2, fifth 1 point. This extra credit is on top of the regular extra credit so one can theoretically get 115 points for this homework project.