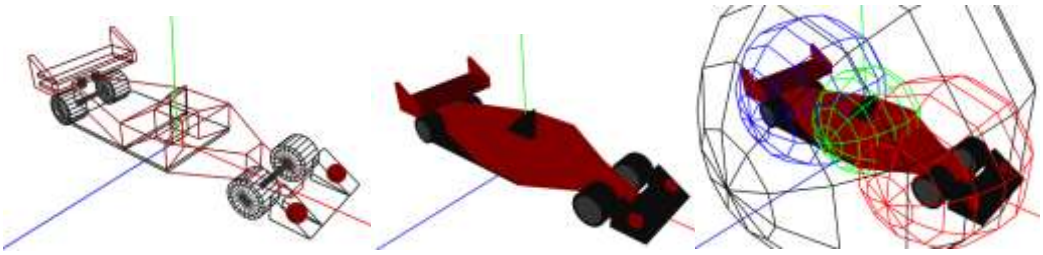
Exercise 5 – Model

Due to 21st of June, 2020 (23:55 Israel Local Time)



Overview

Professor Speed from the **International Driving Center (IDC)** asked Mr. Lazy to model a racing car for him. The car is expected to compete in the next Formula 1 racing competition. Mr. Lazy started working on the assignment, but he got lazy and needs your help finishing the job, otherwise he will get fired.

In this exercise you will practice basic OpenGL techniques in modeling. Your goal is to complete the car model, and extend it with new decorating features. You will also need to support bounding volumes - this will help us in the next assignment when we support collision.

A reference executable can be downloaded from Moodle. Note that you will need to place it in a folder along with the JOGL dlls (Windows) or jnilibs (OS X).

Usage

- The user can rotate the view around the model by dragging the mouse over the canvas.
- Mouse wheel is used to zoom in and out.
- The user can change the window's size and alter its ratio, without distorting the image.
- Pressing 'p' toggles wireframe and filled polygon modes. Make sure to use the wireframe mode - it helps better understand the parts that make each car part.
- Pressing 'm' displays the next model.
- Pressing 'a' turns the xyz axis on/off.
- Pressing 'b' will show/hide the bounding volumes of the car.
- Pressing 'Esc' will exit.
- The application receives no command line parameters.

Modeling

There are endless ways to model a F1 car. Here we require a minimum level of details. You may however embellish the model with as many additional details you see fit.

Building-Blocks

The minimum set of primitives you need to use for modeling are:

- **SkewedBox (Implemented)**: This primitive is already implemented. It will be our main building block, you need to understand how it works, and how each skewedbox is rendered. Notice how the base of the skewed-box is always on the xz-plane. See details in the provided sketch presentation.
- **GLU quadrics -** more information about GLU can be found in the recitations (Rec 8).
    - Sphere
    - Cylinder
    - Disk

Everything in the above image can be obtained using these primitives along with affine transformations. Again, you may use any additional drawing method you see fit.

Structure

Your model must be modular and constructed in a recursive manner using OpenGL's matrix stack. Following is a list of the parts in our model. You need to use the building blocks mentioned above, in order to complete the model.

- F1Car – This is the whole car we want to render. It consists of the following main three parts:
    - Back.java (Implemented) – The back body of the car. This part of the car is implemented, and can be used for reference when working other parts (note that this part renders a PairOfWheels, which still need to be implemented by you). This part is built from:
        - Spolier.java– A spoiler for the car which consists of rods and skewed boxes.
        - PairOfWheels.java – Two wheels connected with a rod.
    - Center.java (implemented) - The center of the car body - It is built from skewed boxes only. Notice that some of the skewed-boxes need to be rotated in order to obtain the desired model.
    - Front.java **(Partially Implemented)** - The front of the car body - It is built from:
        - FrontBumper.java **(NOT Implemented)** - The front bumper is build from three skewed boxes (a wing and base bumper skewed box). The bumper also contain two spheres representing the car front lights. You need to implement this class.
        - PairOfWheels.java - Two wheel connected with a rod (this matches the back pair of wheels).

        - FrontHood.java **(implemented)** - The hood is split into two parts which are skewed-boxes connected to each other. The front also has a bumper, which consists of three skewed boxes.
- Add new design features to the car, use your imagination (**NOT Implemented)**. There are three possibilities, we expect you to implement at least one, but for each additional requirement you will receive **2 points bonus** (so up to 4 points in total):
    - Add Line strips that decorates the car (see here for example).
    - Add exhaust pipes to the back of the car - Place them in a reasonable location.
    - Any other decorations that you see fit (as long as it is impressive)

To make life easier, Mr. Lazy wrote down a list of parameters that can be used when you model the car. These parameters are part of the Specification.java class, and it contains details about the Height, Depth, and Width of each element in the car. The naming convention used is as follow:

- Length is measured relative to the x-axis (Red axis)
- Height is measured relative to the y-axis (Green axis)
- Depth is measured relative to the z-axis (Blue axis)

Make sure that you create the minimum amount of objects required to render each part. For example; you don't need two Wheel objects in order to render the wheels of the front car. Use one Wheel object along with Model transformation.

Symmetry

Our model is symmetric along the z-axis, meaning the car is symmetric about the xy plane. Make sure to model your car such that all elements are symmetric about the xy-plane. For example, when you render a skew-box, the depth along the z-axis and the negative z-axis should be equal - this will make life easier for you.

Bounding Spheres - IIntersectable.java (Not implemented)

In next assignment, we will use our F1 car in a racing game. The car objective is to avoid obstacles within the scene. For this, we need to determine if the car intersects the objects within the scene. This can be done by bounding our car with spheres, and check sphere-sphere intersection (more on this in HW6). For this, we will use a very -simple- Hierarchal bounding volume data structure. The hierarchy is represented as a list of 4 spheres. The first sphere is the main sphere that bounds the whole car, the other 3 spheres are the spheres that bound the front, center and back car body (respectively - in that order). You need to make sure that invoking IIntersectable.getBoundingSpheres will return this structure.

The spheres themselves are represented in the edu.cg.models.BoundingSpheres, and you need to implement this class fully so that we can change the sphere position in case the car moves. We want also to be able to visualize the spheres by invoking the render method.

(10 pts) Bonus

Change the getBoundingSpheres so that the returned value is not a List of BoundingSpheres, but a tree structured hierarchical bounding spheres. The hierarchy is determined as follows:

the root is the sphere that bounds the whole car. The structure of the tree should be in such way that each node's children are spheres within the current node's sphere. The implemented hierarchy should be more fined- meaning, you need to add more levels, covering smaller parts of the car.

Viewing - Implemented

The user should be able to rotate the model using the mouse. The viewing method you **(which is already implement)** is called "Virtual Trackball" and is discussed in Appendix A. Basically it involves projecting the mouse before dragging and after dragging positions on a sphere, and then computing the rotation needed to transform between the two. This transformation then should be performed on the model.

Zoom - Implemented

Zoom should be achieved by **moving the camera** closer to the model. Note that this is possible only when a perspective projection is used.

Lighting - (Not for this Assignment)

In this exercise you can use glColor to set a uniform color for each face or block. You can also set the color using some predefined colored in Materials class. Note, in this class we also define some other properties (such as diffuse reflection etc.), but they are ignored by OpenGL because we disable lighting for this exercise.

Projection - Implemented

We apply perspective projection to render the scene. Note how we apply the required transformations for the model to be displayed in the center of the window, in an appropriate scale.

Additional requirements

Back face culling should remain enabled at all times. This will make polygons transparent from their back side. You may though alter the definition of the back side (GL_CW/GL_CCW).

Recommended Milestones

Write incrementally. We suggest the following implementation milestones:

1. Import and run the given code. Go through it. Read the TODOs.
2. Implement the FrontBumper model in it's own coordinate system. Use the Specification class in order to understand the exact dimensions of the class
3. Implement the Sphere Class without accurate radius!!
4. Implement the getBoundingSpheres for each car part (front, center and back). The returned sphere is relative to each part of the coordinate system.
5. Implement the getBoundingSpheres for the whole car. Simply create a bounding sphere for the whole car, and add the bounding spheres for each of the subparts

(front, center and back) in the output list. Make sure you translate the spheres in order to place their centers relative to the Car Model coordinate system.

Tips

- During design, it might be easier to use an orthographic projection.
- Remember that polygons have to be convex to work with OpenGL.
- There are some places in the supplied code that are meant for ex6. You can ignore them for now.

Submission

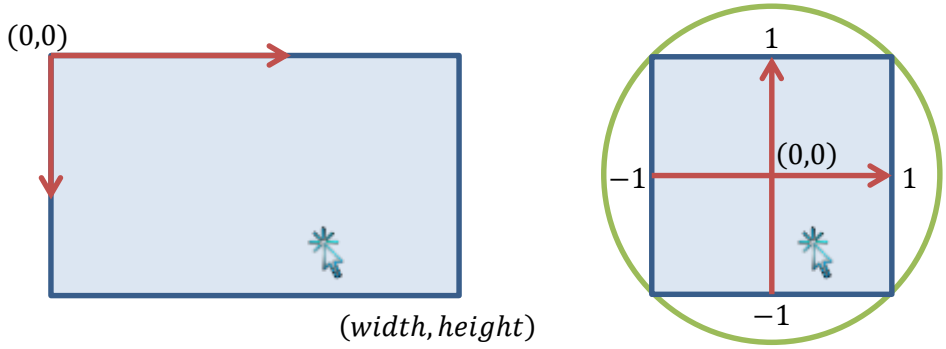
- Submission is in pairs
- Zip file should include
    - All the java source files, including the files you didn't change in a ZIP named "ex5-src.zip".
    - Compiled runnable JAR file named "ex5.jar"
        - This JAR should run after we place JOGL files in the same directory as the jar file.
        - Make sure the JAR doesn't depend on absolute paths – test it on another machine before submitting
        - **Points will be taken off if the JAR fails to run!**
    - A short readme document where you can **briefly** discuss your implementation choices
        - This include new design features you added.
        - The tree structured bounding volume hierarchy (bonus)
- Zip file should be submitted to Moodle.
- Name it:
    - <Ex##> <FirstName1> <FamilyName1> <ID1> <FirstName2> <FamilyName2> <ID2>
- Before submission be sure to check for updates on moodle.

Appendix A – Virtual Trackball

The following is a short description of the trackball mechanism you need to implement.

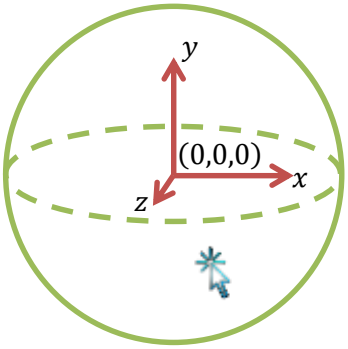
Step 1 – Transform Canvas Coordinates to View Plane

Given a 2D point on the canvas we need to find its projection on a sphere. First convert the 2D canvas point to a 2D point on a viewing plane contained in the sphere. This is accomplished by: $x = \frac{2p_x}{width} - 1, y = 1 - \frac{2p_y}{height}$



Step 2 – Project View Plane Coordinate onto Sphere

Given the view plane's 2D coordinates compute their spherical z-value by substituting them in the sphere's formula: $z = \sqrt{2 - x^2 - y^2}$ (make sure that $2 - x^2 - y^2 \geq 0$).



Step 3 – Compute Rotation

Given the current and previous 3D vectors we need to find a rotation transformation that rotates between the two. A rotation is defined using a rotation axis and rotation angle. Use vector calculus to obtain these. Note: pay attention to degrees/radians.

Step 4 – Rotate Model

Rotate the world about the origin using the ModelView matrix. Note that the rotation is cumulative, so it would be easier for you to store the rotation matrix between redraws. Order of the rotations matters. Denoting the cumulated (stored) rotation R_s and the newly calculated rotation R_n, the new rotation matrix should be $R_n R_s$ (meaning that you should first call glRotate(R_n), and then glRotate(R_s)).