Ex3 - OpenGL Game

TAU, Computer Graphics, 0368.3014, semester A 2009/2010

For this exercise you will implement a complete 3D game and render it using OpenGL. The purpose of the exercise is not only to practice OpenGL but to also learn many computer graphics related aspects such as orientation in a 3D space, Modeling, Parametrization, Collision detection, texturing, Animation etc... It is hard to say what is and what isn't graphics in computer games. Aspects clearly not related to graphics (e.g. Sound, AI) are not addressed (although you may want to).

The exercise is divided into two parts - the first part concerns the creation of an ad-hoc 3d game engine and the second part is about developing an actual game to use that engine. The first part mostly follows strict guidelines and the second part gives more creative freedom.

Part 1 - Basic game engine 2
- Overview 2
- Game states 2
    - Planet state 2
    - Map state 3
- Planets 3
  - Planet motion 5
  - Sphere parametrization 5
  - Planet mesh 6
  - Positioning on a sphere 6
    - Positioning on a sphere with terrain 7
  - Walking on a sphere 7
    - Walking on a sphere with terrain 7
- Rendering 8
  - Lights and shadows 8
  - Meshes 8
  - Skybox 9
  - Planet styles 9
    - Textures 9
    - Color 9
    - Material 10
    - Partial transparency 10
    - Environment 10
  - Portals 10
  - Acceleration 10
- Program flow 11
    - Game loop 11

Part 2 - Customized game play 11
- Gameplay schemes 11
- Advanced Effects 12

Development Plan 12
- Milestones 12
- Tips 13

Logistics 14
- Grading 14
  - Submission 14

Implementation 15
Startup code 15
More useful links 0
Resources 0
Installing JOGL 15
Faq and Updates 16

Part 1 - Basic game engine

Overview

The world in which the game takes place is a constellation a planets. The constellation is composed of numerous orbiting planets. Each planet's orbits around a predefined axis. Each planet has it's own distinct appearance and surface details.

The player controls an actor that is able to walk on planets. Each planet has at least one portal object. When an actor walks into a portal he is immediately teleported to another portal, possibly on anther planet. That way the player can move from one planet to another. By default the player successfully completes a level by reaching a terminal portal (again, the exact game play details are for you to decide, in part2).

In most games the game world is flat and gravity is constant. In this game the world is composed of sphere-like objects, each with a center of gravity of it's own. One popular game that uses this principle is "Super Mario Galaxy" (http://www.youtube.com/watch?v=U-Qw1ClCVN8). Your challenge for part one is therefore to create a game engine that simulates this special environment.

The following sections give more details about what you are required to implement.

Game states

You need to support at least two game states:

1. Planet state
2. Map state

Pressing **m** toggles between these two states.

Planet state

The player controls an actor which walks around on a planet's surface and interacts with his environment. The view in this state is third-person. That is the camera should track the actor from behind at a fixed distance. Upon entering a portal the player is teleported to another portal.

The player controls the actor's movement using the **arrow keys**: Walking forward and backwards with Up and Down arrows, and turning in place with Right and Left arrows.

Map state

In this state the player can view and inspect the planet on which the actor is situated. While in that state the game is still running and the planets are orbiting. The player can switch to this state at any time.

You should implement a basic 3D viewer. The player should be able to rotate the camera around the planet on which the actor is located. Control the rotation using **arrow keys**. Allow to zoom in and out using **+**/**-**. Note that if the planet is moving/spinning then the camera should move accordingly.

Planets

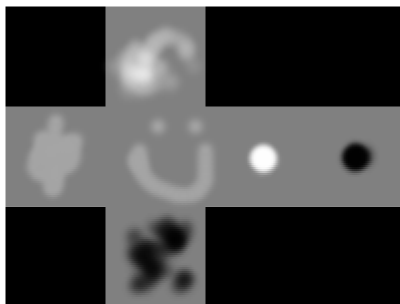
The shape of a planet is essentially a sphere but it can be much more than that: By associating a height value with each point on the sphere's surface we can have a diverse terrain with hills and valleys. To do that you first need to have a 2D parameterization of the sphere's surface (i.e. mapping surface points to 2D). Once you have that you can use a single channel image to define the height value in each discrete parameterization coordinate.

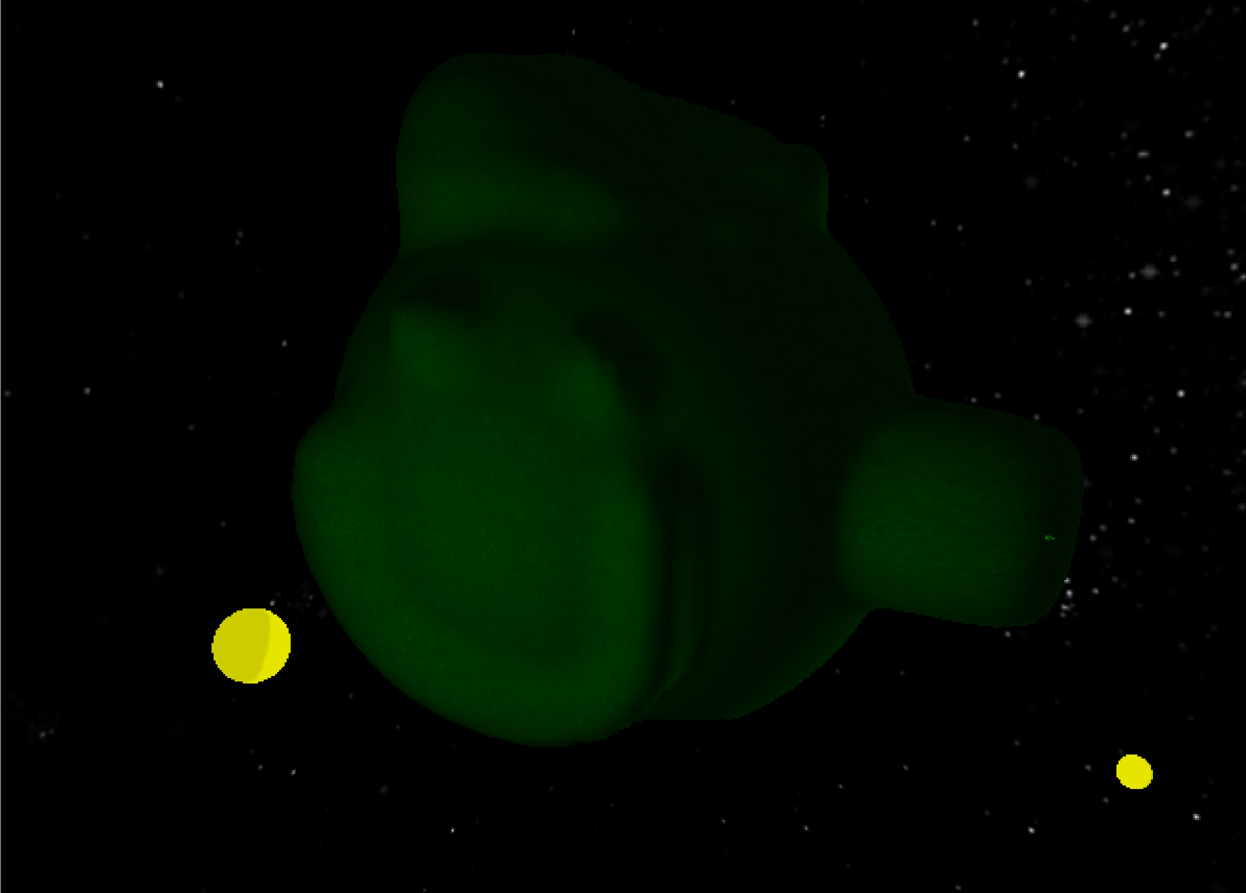
This kind of terrain is called a 2.5D terrain. It's not fully 3D terrain because, for example, you can't use it to describe caves. Many classic games used 2.5D terrains to simplify their logic, map design and hasten critical computations such as collision detection.

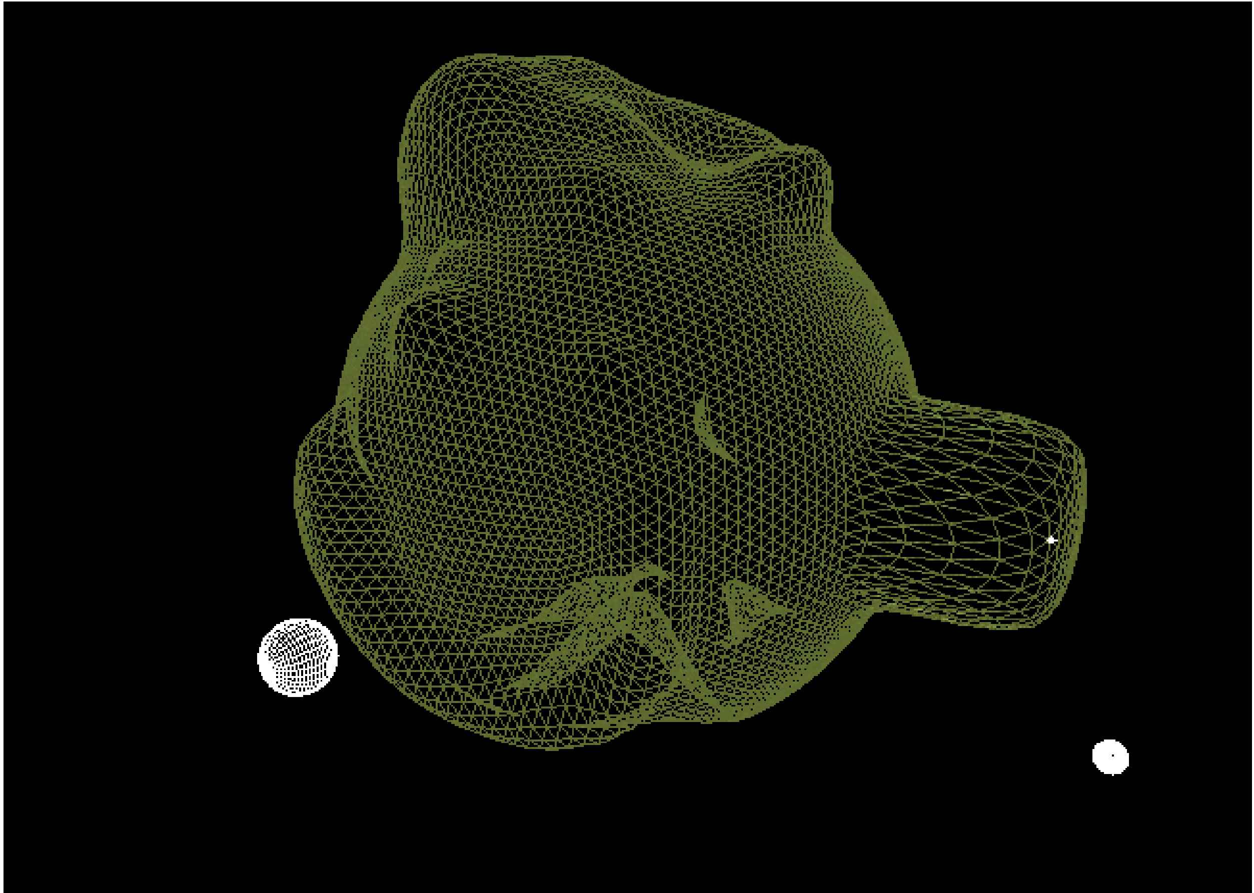
Implementing 2.5D terrain on a sphere is more challenging than on a plane. Creating the initial mesh however is easy: Take a regular sphere mesh and scale each vertex such that it's norm equals it's suitable height value.

The height map is simply a gray-scale image that you can create offline in any photo editing tools such as Photoshop or Gimp, and then load it at runtime before the level starts to generate the mesh.

The following shows an exaggerated example of this. The small picture is a grayscale bitmap file representing the "Height" map. The two larger picture shows a screenshot of a spherical planet distorted by the height map.







You can assume that all objects on a planet are it's children. That is all transformations on the planet effect the objects as well. Objects are not logically effected by planets other than their own.

Planet motion

Planets should slowly orbit in space around a predefined axis. This axis can be either fixed or the center of another planet (e.g. like a moon).

Planets may also rotate around themselves. However this might disorient the player to much and is left up for you to decide.

There should be absolutely no consideration to the laws of physics. You should position and animate the planets by aesthetics alone.

Sphere parametrization

There are many ways to parametrize a sphere. You've already seen in ex2 how to map a sphere using Spherical Coordinates (latitude & longitude). However this method does very poorly for us because it distributes very unevenly on the surface, and it's unintuitive to draw/visualize in 2D.

In Cubic mapping each point on a sphere is projected onto one of the 6 faces of a cube. This allows us to easy draw/visualize in 2D the appearance of each of the faces of the sphere. It still distribute unevenly but it is many times more suitable for our needs than Spherical Coordinates (note that our planets don't have specific poles which we can ignore).

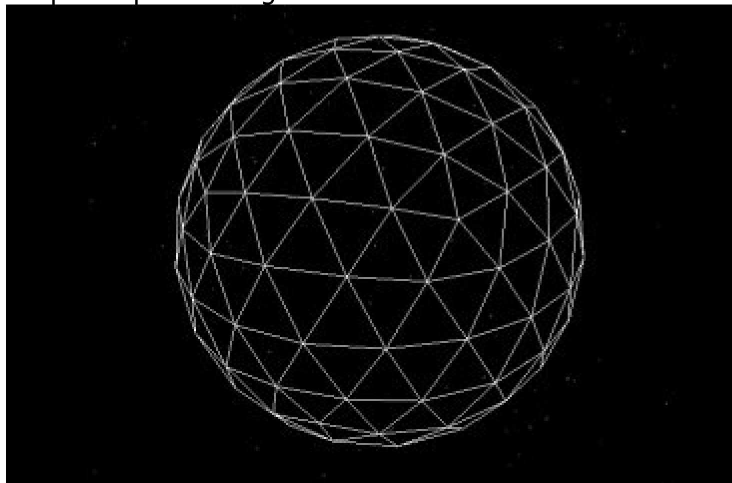
We recommend you use Cubic mapping due to it's all around simplicity, but there might be better techniques for you to use.

Planet mesh

Given a parametrization and height image it is very to generate the planet's mesh; Simply make the norm of every vertex to be equal to the height at it's parametrization.

There are many ways to represent a sphere using a mesh. The regular 3D sphere mesh which is built from slices and stacks like the one GLU makes is very unsuitable for us.

A recursively subdivided Icosahedron, sometimes termed a Geo-sphere, is much better. A Geosphere is composed from equal sized evenly distributed triangles. It can be created very easily by recursively tessellating every triangle into 4 triangles. See tutorial here. The two screen shots above depict a planet originated as a tessellated Icosahedron.



Positioning on a sphere

Walking and positioning on a sphere can be a bit confusing at first. Here I will attempt to guide you through the principles. We assume that the center of the sphere is positioned at (0,0,0).

What transformation places an object on a sphere?

As you know any transformation can be described using 4 vectors: Left, Up, Direction, Location (see tips).

You can assume that objects in your game are never tilted, that is they are always standing up straight. Therefore for a given location on a sphere the *Up* vector is automatically determined:
up = normalized location
In other words anywhere you are in space you can tell that Down always points to the

center of the sphere, and up to the opposite direction.

The *Up* vector defines a circle on which both Left and Direction can reside. Thus given a location, to completely determine the transformation you just need to fix either *Left* or *Direction* (as having one will tell us about the other, by taking cross-product with *Up*).

And that's it. You can now position an object on a sphere given location and direction, or location and left.

It might be more convenient for you to define an objects position using a location and a POI (point of interest). In this scenario all you need to do is:

1. Calculate *Up*
2. Take the cross-product between *(POI - Location)* and *Up* to obtain *Left*
3. Take the cross-product between *Left* and *Up* to obtain *Direction*

Positioning on a sphere with terrain

To place the object on the surface with 2.5D terrain you should

1. Project *Location* to the parameterization space
2. Obtain the Height at that parameter
3. *New Location = Height * Up*

Walking on a sphere

The minimum you need to support for walking is: moving forward, backward, and turning in place.

Using the positioning from the previous section it's easy to see that:

- Moving forward/backward is simply rotating the object around the *Left* vector.
- Turning is done by rotating the object around the *Up* vector.

However moving forward/backward by rotating around a vector is hard to work with. For example the degree of rotation needs to be adjusted with the height in which the moving takes place.

A more practical approach is to approximate the arc movement with a short line movement. That is, to walk forward simply compute
*Est Location = Location + Direction*velocity*

Then normalize *Est Location* to obtain the new *Up*, and use it to compute the new *location*. Since walking doesn't change the *Left* vector all that is left is to calculate the new *Direction* (Cross-product).

Walking on a sphere with terrain

By simply positioning *Est Location* on a sphere with 2.5D terrain you can get very nice results. The object will simply follow the contour of the terrain.

This will seem unrealistic because of gravity consideration. It is outside the scope of this exercise to deal with the physics involved. However simple logic can be used to handle common pitfalls. For example you can prevent walking if the height value in the new

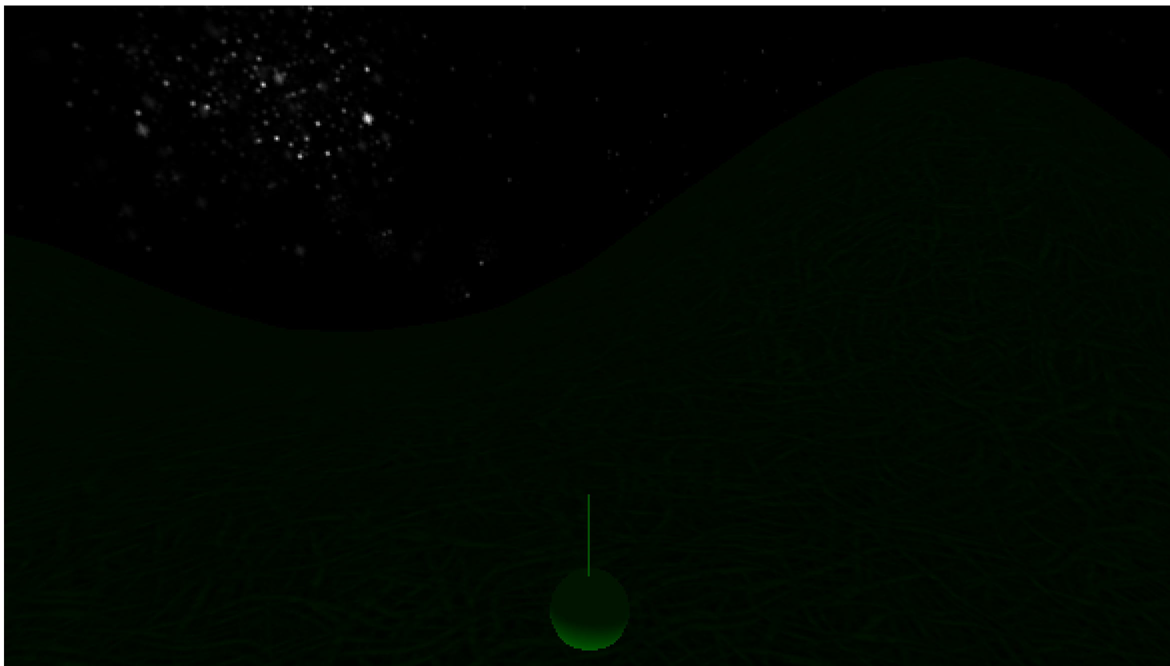
location is much higher than the current height. Again, this doesn't really prevent the actor from climbing steep walls, but you are not required to supply a complete solution.

Rendering

Lights and shadows

Some planets should emit lights that affect all other planet (e.g. suns). Shadows are not required.

There is one situation that you should handle: Say an object is situated on a dark side of a planet (i.e. when the light is on the other side). Unless you do something that object still receives light (i.e. all it's polygons facing the light are lit). It looks very unrealistic to have a partly lit object in a dark environment. The following picture gives an example of this:



Thus if a planet eclipse it's child object from a light source you should turn that light source off when rendering the child. You can use the code for sphere-line intersection from ex2.

Note: This feature is perhaps the single most important feature in this exercise as it the only thing that you can't do with a general purpose scene graph framework.

Meshes

You need to support loading and rendering meshes. You can use the same mesh format as for ex2. We don't require texturing meshes, just regular shading. Meshes should be rendered smoothly like in ex2.

Pay close attention to the resolution of the models and meshes. A model that is going to be small on the screen doesn't need to have thousands of faces. this will just make your game

slow and memory hungry.

For the basic game you only need meshes for the player's actor and portals. Animating a mesh is out of the scope of this exercise and therefore to make walking look good you might want to pick a mesh without legs (ball, car, robot).

Note: A cheap way to animate walking is to edit the mesh and create two meshes in different walking positions. Then in run-time you simply alternate between them.

Skybox

To achieve realism you should also render the world outside the constellation. This is obviously done with textures. One way to do this is with a Skybox.

A skybox is composed of 6 images obtained from an offline projection of a 3D world onto each face of a cube (e.g. using raytracing). Then in real-time we set each image as a texture on a world cube (a huge cube containing most of our world). Thus when standing in the middle of the cube we feel that we see the projected 3D world. You should make sure that the cube is

1. Always centered at the camera (but never rotated)
2. Never writes to the depth buffer so to never hide anything else
3. Never accept lighting effect, especially diffuse and specular

Planet styles

Each planet should have it's distinct aesthetic style apart from it's shape and terrain. Each planet should have it's own color material and rendering themes. That way you could distinguish between different planets and relate portals to their planets more easily. Luckily OpenGL makes it very easy to quickly diversify the drawing style of existing meshes and textures. The following is a list of suggested techniques you might want to play with in order to achieve diversity. You will not be graded according to how many of these you actually incorporate in your game, but instead on the overall aesthetic appearance and diversity of the planets.

Textures

Each planet should have at least one type of texture to define it's terrain. Either rocky, grassy sand etc. Texturing the planet should again be somewhat simple with an existing parametrization. A terrain texture is used as a repeated tile (e.g. here) and good terrain textures are designed in such a way that the tiling is seamless.

Color

The most basic feature that each planet should have is it's own color scheme. It is very easy to alter colors with OpenGL. If you want a purple planet you don't have to use a purple texture. Instead you can use texture blending to incorportate the shading color with the texture and secondly use colored light. Note that while rendering a given planet you can change light colors of global lights (thus for example simulating atmosphere and evironment lighting). The latter is very important, as for example you expect objects on a red planet to appear reddish.

Material

Naturally choosing different values of diffuse and specular factor may have a big effect on a planet's surface appearance from far away. You might also go as far as to calculate normals differently for each planet. For instance a very rocky planet might look better with flatter shading.

Partial transparency

As we've discussed in class OpenGL can use blending to give an illusion of transparency. Thus you can make a crystal-like planet quite easily.

On a closer look when walking on the surface, blending could help give an effect of only partly transparent surfaces - for example an ice floor. You can achieve this look by using blending and **fog** when rendering the surface.

Environment

In addition to giving each planet a distinct look from far away you should also work to make each planet have a distinct feeling when on it. We've already mentioned the importance of color lighting for realism.

Fog is a very simple but efficient OpenGL technique for adding realism through mist and haze, but also to cover up the rigid polygonal appearance by adding noise to the scene.

Things also look differently when looking to space when on a planet. You might want to try and add atmosphere effects by either using fog or even altering the color of the skybox.

Portals

A portal should look like an open doorway. Even two pillars to indicate it are enough. For the interior itself you can use a variety of effects (e.g. see Advanced effects).

In general the portal should appear visually like the planet to which it teleports (for example it should inherit it's material).

Acceleration

You must always use at least one OpenGL acceleration method (either Display lists or vector arrays).

If performance is still an issue then you should code some simple logic to decide when or not to render expensive things. For instance you can refrain from rendering items on planets other than the one you are on.

In addition you can render non-current planets using fewer vertices. You can easily accomplish this since your able to control the level of tessellation of a planet. You can even use downscaling from ex1 to obtain a smaller height map that would require less vertices to approximate.

Program flow

While you're not required to provide a game menu you might want to at least implement one in GUI. After starting the game you should load the first level. When finished with win load the next level. When lose restart the level.

Game loop

You can implement any game loop you desire. The only important thing is that game updates occur in constant intervals.

Part 2 - Customized game play

Gameplay schemes

Now that you have the basic ad-hoc game engine you can easily use it to make a fun and captivating game. For this part you are pretty much on your own but you are still required to implement a minimum set of features. The following is a short list of possible game styles. You should choose one and fill the minimum requirements. It's possible to work on your own idea but be sure to first verify with chen.goldberg+TAU_CG@gmail.com.

- Arcade style
  - Player quickly runs through the planet gathering coins and earning credit. Avoid collision with dumb enemies. Kills enemies by jumping on them.
    - Jumping - No need for physics calculation or anything. Just a 2.5D jump.
    - Gathering coins and trophies
    - Dumb Enemies that runs toward the actor. Upon collision player loses life.
  - Like
    - Any Super Mario game
- Shooter style
  - Player walks around, killing his enemies and accumulating new weapons and ammo.
    - Gather weapons and ammo
    - Use multiple weapons
    - Point projectiles*****
    - Dumb enemies that fire at you
- Driving style
  - Player drives though the planets, race against time. Planets should be designed as long race tracks.
    - Car-like control - Moves even without input
    - Gathering Power-ups
    - Better physics - allow car to tilt and respond to the normal of the terrain.
- Sport style - Golf
  - Player plays Golf. Aims ball and shoots. Where ever the ball stops, actor starts again. Instead of a hole the ball should hit a portal which will teleport it.

        - Ray projectile*****
        - Ball physics - Ball should bounce (reflection) and roll.
- Worms style
    - Turn based game with multiple actors. Actor can stop and aim a rocket. Fires projectiles at enemies on same and other planets. Wins when all enemies are dead.
        - Ray projectile***** - Rockets should be effected by gravity of all planets (approximation).
        - Destructible terrain (easy in 2.5D!)
    - Like
        - Worms3D, Scorched Earth
- Puzzle style
    - Each launcher is locked by a key which a player must get before moving on
        - Something like portal, braid etc.

* Some games require different kinds of projectile collision detection. There are two kinds:

1. Point projectiles - Collision between this projectile and an object/terrain is performed only at the location of the projectile. That is, it requires a Point collision detection which is very efficient. Thus games that use this kind can have many projectiles at the same time. This limitation forces either projectiles to move slower or have (fatter enemies)
2. Ray projectiles - Collision between this projectile and an object/terrain is performed at the path of a projectile over a turn. Thus allowing very fast projectiles. Since we don't want you to spend time optimizing this, only games that use a single projectiles at the same time should implement this. Generally it shouldn't differ much for what you've implemented in ex2.

Note: If you implement the basic game engine sufficiently well then this part should be rather simple. All it takes is just a small amount of game logic that operates the existing scene graph, and a few specific engine modifications.

Advanced Effects

Implementing any of these advanced OpenGL techniques will give you a considerable bonus.

- Custom shader effects
- Shadows - Shadow volumes or shadow maps. If you implement shadows then it's Ok if you don't animate the planets.
- See through Portals - The interior of a portal is a dynamic texture showing what's on the other side of the portal (e.g. here) , as if the destination is on the other side of the portal. Requires rendering to texture. Or playing with stencil, depth buffers.
- Particle systems

Development Plan

Milestones

The following is a suggested breakdown of the development process to short term goals

- Rendering the constellation

    - First create the scene graph classes and their renderers
    - Render the planets in space. For now use a dummy model to visualize them (e.g. gluQuadric)
    - Let some planets emit light and enable lighting
    - Allow the user to control the camera and view the scene around a fixed location
    - Animate the planets
- Positioning an object
    - Now focus the view on a single planet fixed at the center of the axes
    - Position and render the actor on the sphere's surface
    - Allow the player to control the actor (Walk, turn)
- Rendering a planet
    - Replace the dummy planet models with your own sphere mesh
        - Render smooth mesh
        - Acceleration
    - Implement parametrization
    - Load height map
    - Draw planet with 2.5D terrain
    - Position and move actor according to 2.5D terrain
    - Implement Planet State and let player follow actor
    - Apply texture to planet
    - Add portal objects
- Tying it all up
    - Allow actor's planet to animate
        - Make sure everything on planet works
    - Activate portals
    - Implement a SkyBox
    - Handle the "dark side" effect
- Customized game play
    - Consider what each game style requires you to add to the engine and decide which is most suitable for you.

Tips

- Some code from ex2 and even ex1 is relevant here! Be sure to use it.
- You need a good vertex/matrix library. Java3D has some nice features.
- I can't stress enough how important it is to adopt an object oriented approach to this exercise. Cramping all the code in a single class won't do it this time.
- Using OpenGL glRotate and glTranslate to position objects in space is nice but rotating will drive you insane when dealing with spheres. A saner approach is to use glMultMatrix and supply your own transformation matrix:

$$\begin{pmatrix} left.x & up.x & direction.x & location.x \\ left.y & up.y & direction.y & location.y \\ left.z & up.z & direction.z & location.z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Terrain glitches will happen and it's understandable. For instance walking next to a steep hill, an actor might overlap the terrain. This is because we use a different model for rendering and for logic. We render using a sampling of the height map, thus if we don't sample densely enough then have these glitches. Again it's understandable.

- Terrain texture glitches also might occur, since the process of mapping a 2D image onto a sphere always creates problems around the edges. It is understandable but should be kept as hidden as possible.
- A height map should be a grayscale image with values 0-255. Thus it only gives you a relative height factor [0..1]. To convert it into absolute height you should add a parameter for each planet determining the range of elevation.
- Using parametrization and maps you can describe the surface of the planets in far more details. For instance create a bitmap for specifying terrain texture in each location (e.g. areas of the image with color 1 represent grassy areas, 2 represent stone areas...). Also with parametrization you can design planets offline by specifying locations of objects on the planet.
- In positioning, remember that the order of vectors in cross-product matter, so be consistent.
- Don't use Java timers!

Logistics

Grading

- Basic game engine - 55
- Custom Game play - 35
- Performance and aesthetics - 10
- Advanced graphics - Bonus

Performance is important in a real time game. At least 30fps on a modern desktop machine with a modern graphics card. We are aware however that compatibility is always an issue, and while a game may run flawlessly one platform it might crawl on another. We take this in mind.

Submission

As before submission is in Pairs.
In the due date you need to submit a **single** zip file with the following:

- Full source code
- JAR file.
- A short document (2-4 pages) explaining:
    - Instructions for the game
    - Features you implemented
    - The structure and design of your code
    - Anything else needed to make it work. Please supply

Please submit your work to **chen.goldberg+TAU_CG@gmail.com**

Send One email containing everything you submit and wait for a confirmation. if you don't receive a confirmation response after two days send again.

Please make sure you read the "FAQ and Updates" at the bottom of this page frequently and before your submission.

Implementation

We recommend you work with Java and JOGL for OpenGL support. You are welcome to implement the game in C++/C#, but we can't guarantee support.

Startup code

This file is an initial framework for OpenGL. It contains the basic entry points discussed in class and a simple scene made of two polygons so you could see that OpenGL works. feel free to change this file anyway you like.
Main.java

Links

- SWT/JOGL
    - http://www.eclipse.org/articles/Article-SWT-OpenGL/opengl.html
    - http://kenai.com/projects/jogl/pages/Home
- Documentation
    - http://download.java.net/media/jogl/jogl-2.x-docs/
    - http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/opengl/package-summary.html
- A nice Tutorial to get you started
    - http://www.geofx.com/html/OpenGL_Eclipse/OpenGL_Eclipse.html
    - Notice that this tutorial talks about an *Eclipse plugin* and not a stand-alone applications so you will need to ignore some of the things mentioned
- This guy wrote many books about developing games in Java, Java3D and JOGL
    - https://fivedots.coe.psu.ac.th/~ad/index.html
- OFF meshes
    - http://shape.cs.princeton.edu/benchmark/
    - http://shapes.aimatshape.net/viewmodels.php?page=1
- space skyboxes I found
    - http://quadropolis.us/node/2019
    - http://gfx.quakeworld.nu/details/266/space/
    - Google for more
- Textures - many free terrain textures online

Installing JOGL

Windows:

1. download JOGL from:
   http://download.java.net/media/jogl/builds/archive/jsr-231-2.0-beta10/jogl-2.0-windows-i586.zip
   Don't be tempted to download the AMD64 version. It will cause you nothing but trouble.
2. Save the zip file is your directory of choice, for instance in "**C:\Program Files\java**"
3. Extract the zip file into a directory - "**C:\Program Files\java\jogl-2.0-windows-i586**"
4. Create a new project in eclipse and add the startup code supplied below - Main.java.
5. Right click the project, go to properties->Java Build Path

6. Click "Add External JARs" and select the two JARs under "**C:\Program Files\java\jogl-2.0-windows-i586\lib**"
7. Do the same to add SWT.jar from wherever you installed it.
   After this the startup code should be able to compile but when you try to run it, it will likely fail with an exception. the JOGL DLLs are still missing.
8. Locate the directory of the JRE or JDK you are using. for instance "**C:\Program Files\java\jre1.6.0_02**". you can do that using the properties->Java Build Path dialog. look under the JRE to see where it is installed.
9. Copy the JOGL DLLs from "**C:\Program Files\java\jogl-2.0-windows-i586\lib**" to the \bin directory under of the JRE, in this example:"**C:\Program Files\java\jre1.6.0_02\bin**"
   This is the safest way to make java find these DLLs. there are other methods as well which invlove changing the PATH environment variable. do this at your own risk.

Unix

1. If doesn't work ask me (Chen)

In the end, the DLLs and The JARs in your project should be of the same version. If you have other versions of JOGL installed for some reason, make sure that java uses the right ones.
After completing this the startup code should be working and displaying an OpenGL view.

Faq and Updates

No news is good news!