

# Programming OpenGL

Notes for CSE 470

Instructor: Peter Wonka

## 1 Project Guidelines

- Have fun, learn, be creative, and create something you are proud to show-off!
- No teamwork allowed! It is ok to discuss the projects with other students. You can use the discussion board on myASU (The TA will also try to answer your questions posted at myASU); however your code must be your own. (You would be surprised how easy it is to detect copied code!). You cannot post code on the discussion board! (only very small code sections)
- Comment your code - you will be glad you did. You will be reusing code throughout the semester.
- You will demo each of your projects. In each demo, you will be asked to demonstrate the functionality of the program and possibly describe how some of the code works. Be prepared for this.
- Do not miss your demo! If you cannot make your demo, notify the TA beforehand. If you don't, you will get a penalty of 5%.
- You will upload your projects on myASU. Details about this will be announced in class.
- Late projects will loose points:
  - 7% for up to 24 hours too late.
  - 15% for up to 4 \* 24 hours too late.
  - The myASU timestamp will show when you uploaded your project.
  - 25% for one week late
  - 50% for two weeks late
- If you have **good reasons** that you cannot complete a project on time **and** you have written documentation, then we can make adjustments to due dates. However, you must notify us before the due date that you would like to discuss such an arrangement. Good reasons would be illness, family emergency, visiting a conference to present a paper, ...
- All the University, Student Life and Fulton School academic integrity policies hold: <http://www.asu.edu/studentlife/judicial/integrity.html>

## 2 Getting Started

The projects assume that you are using: Microsoft Windows, Visual C++, and OpenGL

### 2.1 Downloading the Compiler

#### 2.1.1 At Home

For your home computer you can download the compiler from:

<http://www.microsoft.com/downloads/>

Search for: **Visual C++ Toolkit**

Install the toolkit.

If you are a student at CSE, you may get the free Visual Studio from ASU.

### 2.1.2 In the lab

In the lab the compiler should be installed.

## 2.2 Getting your programming environment ready

### 2.2.1 Visual Studio .Net

- Start Visual Studio .NET \* File → New → Project \* Select Visual C++ Projects → Win32 → Win32 Console Project
  - Give your project a name and choose where to store it
  - Click Finish
- Disable support for precompiled headers
  - Remove the '#include "stdafx.h"' line from your project1.cpp
  - Open Project -> projectname Properties (last option)
  - Go to C++ -> Precompiled Headers
  - Change "Create/Use Precompiled Header" field to "Not using precompiled headers"
- Add glut32.lib to your linking settings
  - You should still be in the project settings window
  - Choose Linker -> Input
  - Add 'glut32.lib' to your Additional Dependencies
  - Click OK to return to your main project window

Now we need to load the glut files

- Put the glut32.dll in the same directory as your executable
  - Download from myASU (glut-3.7.6-bin.zip 117 KB). Note: glut is also freely available on many internet sites.
  - When the zip folder opens, copy "glut32.lib", "glut32.dll", and "glut.h" to the directory you stored your project in
  - Now, any time you see '#include <gl/glut.h>' replace it with '#include "glut.h"'
- You should now be able to build your project
- Also, if you paste in code from any of the examples it should work

### 2.2.2 Visual C++ 6.0 (outdated)

The GLUT files go into folders that are mirrored by their OpenGL entries. I would recommend searching for the opengl files mentioned below to find the folders specific to your machine.

1. Your "WINDOWS/system32" folder should already contain "openGL32.dll" and "glu32.dll". Copy the "glut32.dll" here.
2. The "opengl32.lib" will live in a folder named something like "..\Microsoft Visual Studio\VC98\lib". Copy the "glut32.lib" file here.

3. Similarly, the “gl.h” file will live in a folder like “..Microsoft Visual Studio\VC98\include\GL”. Copy “glut.h” here. (Note that “glut.h” defines all the necessary opengl include files, so you only need to include it in your program, instead of including “gl.h” and “glu.h” as well as “glut.h” together)

To start your own program in VC++, do the following.

1. Start VC++
2. File->New->Open a console application
3. Select an "empty project" and pick a name and directory
4. File->New->C++ source (pick a name and directory)
5. You are ready to go! (See a sample program for the basic includes and program structure.)

If you are not allowed to put files in the system and VC folders, then do the following.

1. Put the GLUT files in a folder of your choosing.
2. Open VC++
3. Choose Tools->Options
4. Choose the "Directories" tab.
5. In the *lib*, *include*, and *executable* sections, add the folder in which the GLUT files live.
6. With this set-up, the <GL/glut.h> in your program needs to be changed <glut.h>

### 2.3 Download and Compile two sample programs

To test your installation a first good step is to download and install two sample programs from myASU.

## 3 C and C++

C and C++ are closely related to Java. If you had no previous exposure to either C or C++ you should try reading a tutorial on the internet. You can search for “C for Java programmers” or “C++ for Java programmers” on google.

For OpenGL you will start out using only the C part and then later on use some C++ functionality to structure a more complex program.

Example:

C for Java programmers -

<http://www.cs.cornell.edu/courses/cs414/2001SP/tutorials/cforjava.htm>

C++ for Java programmers –

<http://www.cs.brown.edu/courses/cs123/javatoc.shtml>

## 4 Upload your projects

How to upload a Project

This is a short guide for the successful upload of a project:

- Check the due date of the project on myASU
- Use myASU (digital dropbox) to upload your project. (Do not send the program as email attachment to the instructor or TA! )
- Your project should be packed in a zip file and named with the following naming convention:  
PX.Lastname.V1.zip  
(if you submit some updated files, use a different version)  
So my second project would be named  
P2.Wonka.V1.zip  
If I then found a last minute mistake and want to upload again I would use the name P2.Wonka.V2.zip
- If there is someone with the same last name in the course you can add your first name after the last name in the filename.
- The zip file should include all the source code and should be ready to compile. That means it should include your Visual C++ project files.
- The zip file should include an executable and the glut32.dll so that it can be started directly by clicking on it. (from wherever the file is downloaded to).
- A short text file called “instructions.txt” – this text file should include the following:
  - a) instructions about how to navigate in your program, including the function of special keys (e.g. x exits the program)
  - b) special functions that you implemented
  - c) changes from the original specification that you negotiated with the instructor or TA to the original specification
  - d) other things you consider important
  - e) specify the libraries or code that you used and did not write yourself.You do not need to specify code that we provide on myASU.
- You can use rar instead of zip to compress your files

## 5 Project 1

### 5.1 Purpose

This project has been designed for you to get experience with several basic computer graphics concepts and write a first interactive computer graphics application.

### 5.2 Overview

You should design a simple labyrinth game in which you can control one moving object, e.g. a *car*. The *scene* is viewed from the top looking down. The scene is modeled in three-dimensions.

Note: The first project is the basis for the second project.

Note: You are free to change the interpretation of the project and write another similar application, e.g. a space ship flying through space, a person walking through a city, a person walking through a garden.

Your window will be split in two: the bottom view will be the control panel and the top will show a labyrinth.

You will write an interactive / real-time application where the movement of one object is controlled by user input and the movement of some other objects will be computed along fixed movement paths.

A pop-up menu and keyboard controls (maybe mouse controls) will be used to control the application. See below for details.

### 5.3 Required Functionality

Here are the elements that your program must have. Despite this "laundry list" of requirements, there is a lot of room for creativity! Note: If the text suggests a menu item "Sound - Off" that would mean that "Off" is an entry in the submenu "Sound".

- **10 pts – OpenGL program / Double Buffering:** You created a program that uses OpenGL and GLUT. The program compiles and runs. A window appears on the screen. You use double buffering. Something moves on the screen. The animation is flicker free.
- **10 pts – Popup Menu:** Implement a pop-up menu to control the application. The content of the pop-up menu is described in the following.
- **10 pts - Two Viewports:** - Create two viewports. One for the labyrinth view and one for your control panel. (It seems to make the most sense if your controls are at the bottom of the window, but if you have a different idea, that's ok too.)
- **5 pts - Vector Library:** you should use a vector library. Either download one from the internet, use one provided by us, or implement one yourself. The vector library should be able to compute simple functions on vectors. At least addition and subtraction of two vectors, multiplication and division by a scalar, normalization, vector length computation, and cross product has to be implemented. All vertices (and later normals and texture coordinates) have to be stored as vectors.
- **10 pts - Mesh Data Structure:** You should create a mesh data structure for all elements of the scene. The simplest data structure stores the vertices of a mesh in a dynamic array = an STL-vector. A second STL-vector stores the indices or index triplets. Each triangle is defined by three indices. The indices in the index STL-vector denote the location of the vertices in the vertex STL-vector. A third STL-vector stores the normals. The use of STL is mandatory. For this project you only need vertex positions and vertex colors. In the next project you need to extend with per face normals, per vertex normals, and per vertex texture coordinates.
- **20 pts - Create a Labyrinth:** The Labyrinth is drawn as a collection of objects (meshes). You should at least implement *boxes* and *cylinders* as obstacles and a *ground plane* for them to stand on. The three types of elements, box, cylinder, and

- plane should be generated procedurally. That means you write a respective functions that creates a mesh data structure describing the object( E.g. CreateBox(... ) ). The Labyrinth also needs a *Start Area* (e.g. color box on the ground) and a *Finish Area*. The main idea is to place boxes and cylinders to form walls in the labyrinth. At least 20 wall elements should be placed.
- **10 pts –Object Loader:** Use or implement a file loader to load several unique 3D objects that will appear in the Labyrinth. For example: a car that will eventually be controlled by the user, a street lamp, a person, a park bench, a pedestrian crossing, a mailbox. Unfortunately, this will be limited by what types of models are available. You can load the car as 3D mesh taken from the internet or use a supplied model on blackboard if you want.
  - **10 pts - Car control:** Implement keyboard control for one selected object (e.g. a car). The car is positioned in the Start Area (Box). Use the following special keys: **break (cursor down), accelerate (cursor up), turn left (cursor left), turn right (cursor right)**. To make it harder you can forbid the user to break to speed 0, but always go forward a minimum speed. The program should recognize when the user navigated the car to the Finish Area. If you want you can also improve these simple controls and make a more complex control (e.g. move forward, backward, and control the position of the wheels). Use a pop-up menu (that appears when you right-click the mouse button) with the choices of *quit* or a pick animation speed submenu. In this submenu, allow for slow, medium, and fast.
  - **10 pts – Object rendering:** Write a rendering routine that renders all objects stored in the mesh data structure. The objects should have colors per vertex (can be assigned randomly). The menu items "Shading - Flat" and "Shading - Smooth" should switch between smooth shading and flat shading.
  - **10 pts – Display lists:** In your rendering routine, allow to render objects in intermediate mode (using glBegin() ... ) and using display lists. Use the menu entries "Display List – ON" and "Display List – OFF" to switch between the two modes
  - **10 pts – Text:** There has to be some text displayed in the control panel. An example is the name of the car and the blinking word "FINISH" once the car reaches the Finish Area.
  - **10 pts – Timer:** The timer times the length of the drive (walk, ...). The timer is reset when the car is positioned in the *Start Area*. When the car reaches the *Finish Area* the time stops. The elapsed time is displayed in the control panel during the drive.
  - **10 pts - Moving Objects, Double Orbit:** In addition to the user controlled car moving, other objects should move automatically. You should implement an animation of objects in the labyrinth in a double orbit, e.g. a person rotating around a person which rotates around a mailbox.
  - **10 pts – Moving Objects, Static Orbit:** An object rotates about another one *without* rotating itself. (Consider rotating the upright letter 'A'. A normal orbit would cause the letter to turn on its side, upside down, etc. A static orbit on the other hand will keep the 'A' upright. ) This effect must be made visible by applying it to an object with a coloration or shape variation. For example, if you use a circle it is impossible to figure out the orientation of the circle.

- **10 pts – Reshape Window:** If the user resizes the window, the geometry should not deform! That means the geometry can be scaled in both directions with the same factor, but you cannot scale the geometry in one direction more than in the other. This can result in some empty areas in the window.
- **5pts – Viewing:** Implement two viewing options. One is an orthographic projection from the top, the other one is either an orthographic projection from the lower left looking at a 45 degree angle downwards (similar to many real-time strategy games) or a similar perspective projection. Use the menu options “Viewing – Top Down” and “Viewing – 45” to switch between these two options.
- **10 pts – Design / Make a Game / Effort:**  
We will judge the general “design” aspect of the program. This includes various aspects. E.g. programs with additional features, visual design, ... will get some more points and programs with undesirable design choices will get less.
- **15 pts – Oriented Bounding Box Computation - Graduate Student Requirement:** If you are a graduate student this feature is mandatory. If you are an undergraduate student you do not get points for this feature (except maybe a guaranteed high point value on design). Implement a function that can compute an axis aligned bounding box and another function that can compute an oriented bounding box for any mesh data structure. The menu entries “Bounding Box – OFF”, “Bounding Box – AA”, and “Bounding Box – O” are used to switch the rendering of bounding boxes to either no bounding box, the axis aligned bounding boxes, or the oriented bounding boxes, for all objects in the scene.
- **10 pts – Collision Detection - Graduate Student Requirement:** A collision detection algorithm detects collisions between your car and any other objects in the Labyrinth (This includes the static walls and the moving objects). Implement another routine that can test *axis aligned bounding boxes* (the simple case) for intersections. During runtime simply test the axis aligned bounding box of the car against the axis aligned bounding box of all other objects in the scene. Use the menu entries “Collision – ON” and “Collision – OFF” to turn collision detection on and off. Implement some reasonable behavior in the program for when the collision is detected, e.g. the car bounces off in another direction or the race stops and the user has to start from the beginning.
- **3 pts – Organization:** following instructions from Section 4. ( Naming convention, instructions.txt, program can execute, ...)

## 5.4 Grading

These points represent an initial weighting of the functionality. **Important:** You get full points for a *good* implementation of the required functionality. In principle you will get full points if you implement the requirements correctly, but there are many ways to do things according to specification that we might consider *unreasonable* or *undesired*. Especially we will deduct points if the design choices make it difficult to judge the program (E.g. you choose materials that are so dark that it hard to see anything, the objects move too slowly or too fast, the control of the program is extremely difficult.)

Note: we might change the weights slightly if a requirement is misunderstood by a

majority of the class or some other special circumstance. I do not expect that to happen though.

## 6 Project 2

### 6.1 Purpose

The goal of writing this program is to get experience with:

- Three-dimensional navigation
- Normal calculation
- Lighting
- Textures

### 6.2 Overview

You will add more complex visual appearance to the program implemented for project 1. That includes adding texture images and implementing the phong model for shading surfaces.

For shading and lighting you will need: a) a light source, b) to specify materials for all geometry in the scene, and c) (normalized) normals for all geometry in the scene.

### 6.3 Required Functionality

Here are the elements that your program must have. Despite this "laundry list" of requirements, you can add some other interesting features! Note: If the text suggests a menu item "Light - Off" that means that "Off" is an entry in the submenu "Light".

- **10 pts – OpenGL program / Double Buffering:** You created a program that uses OpenGL. At least one window has to appear on the screen. You should use double buffering. Something moves on the screen. The animation is flicker free.
- **10 pts – Texture Coordinate Generation:** Generate texture coordinates at least for the cylinders, the boxes, and the ground plane. To achieve that you have to extend the procedural generation code from Project one to include texture coordinate generation.
- **10 pts – Texturing:** Apply textures to the scene. A texture is loaded from a file. You can choose between "Texturing - On" and "Texturing - Off" to switch texturing on and off. You should use at least three different textures to texture selected elements of the scene (at least boxes, cylinders, and the ground plane). You should implement **two** different texturing methods for the polygons:
  - In the first version the color of the texture defines the final color of a pixel (of the polygon). ("Texture - Replace")
  - In the second version the color of the texture is modulated (multiplied) with the color computed by the lighting and shading equations. ("Texture - Blend")
- **15 pts – Filtering:** There are two texture filters: One for minification and one for magnification. For both filters you should have a simple and a high quality setting. Use menu entries "Minification - Simple", "Minification - Smooth", "Magnification - Simple", and "Magnification - Smooth".



- **Simple Filtering:** The first version is nearest neighbor filtering (looks bad). Nearest neighboring should be used for the minification and the magnification filter.
- **Smooth Filtering:** The second version is Smooth Filtering. Smooth Filtering requires different settings for the minification and the magnification filter. For the minification filter you will need to generate MipMaps first! For the magnification filter you should just use the filter option that looks the smoothest. The learning objective for this requirement is to understand texture filtering options. You should determine which filter is the best and looks the “smoothest”. In your program you should be able to demonstrate how the texture looks very blocky if you zoom in and have Simple Filtering selected and then how this will be much smoother with the Smooth Filtering selected. Then you should be able to demonstrate how aliasing artifacts occur if you zoom out (with Simple Filtering) and how these aliasing artifacts are diminished when you use Smooth Filtering.
- **5 pts – Expand the Mesh Data Structure:** Your mesh data structure should be able to store per face normals, per vertex normals, and per vertex texture coordinates.
- **10 pts – Per Face Normal Vectors (+ Normal Visualization):** You need to implement a routine that computes per face normal vectors for each triangle in the scene. If the user selects the menu item "Normals - Per Face Normals" the scene is rendered using per face normals.  
If the user selects the menu item "Normal Visualization - On" and the application is in per face normal mode the per face normals should be visualized as lines attached to the middle of the triangles. The normals have to be visualized with some reasonable length so that they are easily visible. "Normal Visualization - Off" turns off normal visualization.
- **15 pts – Basic Per Vertex Normals (+ Normal Visualization):** To make shading smooth, you should compute a per vertex normal for each vertex in the scene. To compute a per vertex normal of a vertex  $v$ , you have to average the per face vertex normals for each triangle that shares the vertex  $v$ . You should implement two versions. One version simply averages the normals and the other version computes a weighted average, so that the normals are weighed according to the area of the incident triangles. (selected by "Normals - Per Vertex Normals" and "Normals - Per Vertex Normals Weighted"). If the user selects the menu item "Normal Visualization - On" and the application is in per vertex normal mode the normals should be drawn as line segments starting at the vertices.
- **15 pts - Crease Angles:** Please note that averaging vertex normals only makes sense for smooth surfaces. Therefore you should be able to specify a crease angle. The crease angle can be either 15, 25, 35 degrees, or off. The options can be selected over the menu entry "Crease Angle - 15", "Crease Angle - 25", "Crease Angle - 35", and "Crease Angle - Off". The meaning of the crease angle is that normals get only averaged at a vertex when the angles between the adjacent triangles is less than the crease angle. The models that you use also have to have some sharper edges so that this calculation makes sense.

- **15 pts – Shading:** The scene should have one single light source (the sun). The light source should be above the scene, but near the edges of the scene. The light source color should have the diffuse and specular components set to (1,1,1). The light source does not use any attenuation (otherwise the scene might become too dark). You should specify materials for all geometry in the scene. There are three options "Materials - Diffuse", "Materials - Specular", and "Materials - Design".
  - If the first option is selected all elements of the scene are rendered with the same material that has a high diffuse and a low emissive component.
  - In the second option all elements are rendered with a material that has a high specular, high diffuse, and low emissive component.
  - In the third option you can assign different materials to different scene elements to make the scene look good.

You might want to have some ambient term or emissive term just to give some basic color and then mainly use the diffuse lighting. A specular term also has to be present, at least for some objects.

The menu items "Shading - Flat" and "Shading - Smooth" should switch between smooth shading and flat shading. You can add other light sources in a different custom mode (not required), but it always has to be possible to analyze the shading with the basic specification (one light source) described above. This is necessary to visually debug your normal calculation and the shading and lighting.

- **Note:** The different navigation modes described in the following do not have to work in concurrently. E.g. in light source navigation mode, the user does not need to be able to control the car.
- **5 pts - Light source navigation mode:**  
If the user selects "Navigation - Light Source" the user can control the light source in the scene. If the user presses the left cursor or right cursor, the light source should rotate around the scene about the Y axis (in circles parallel to the ground plane so that if you press a key long enough you eventually end up at the starting position).
- **10 pts – Navigation1:** This mode is selected by "View - Overview". In the first mode the camera should be above the terrain, near the edge of the terrain. The camera looks at the center of the terrain.  
If the user selects "Navigation - Camera" the camera is controlled as follows:  
If you press the left or right cursor key the camera rotates around the terrain about the up axis (i.e. the camera stays parallel to the plane), the viewing direction is fixed on the center of the terrain however. If you press the up or down key, the camera moves higher above the terrain or closer to the ground. Use the *a* and *s* key to move the camera further away, or closer to the terrain.
- **10 pts – Navigation2:** This mode is selected by "View - First Person". In this mode the camera is located in the car and moves while you navigate the car. This shows the view through the windshield. You do not need to render the car in this navigation mode to avoid occluding the view by the car geometry.  
If the user selects "Navigation - Camera" the camera is controlled as follows:  
You can go forward, backward with the cursor keys or rotate using the left and right keys. Use the *a* and *s* key to control the elevation over the ground plane (a real car cannot do that).

- **20 pts – Procedural Ground Plane Generation - Graduate Student Requirement:** You should generate two different types of ground planes procedurally. The idea is to write a procedure that generates a plane as a regular grid of vertices (probably at least a 500 x 500 grid). Each vertex is then displaced by a height value. Use the menu to choose between “Plane – Flat”, “Plane – Multiscale” and “Plane – Marble” to switch between different types of ground planes. The displacements will be done by Perlin noise and are explained in the slides.
- **3 pts – Organization:** following instructions from Section 4. ( Naming convention, instructions.txt, program can execute, ...)

## 7 Project 3

### 7.1 Overview

The project has two parts. First, you should fix problems from projects 1 and 2. Second, you will read a (simple) research paper and implement it. The second part can be implemented as new separate 2D application using OpenGL to render the textures and glut to manage the menus.

### 7.2 Required Functionality

Here are the elements that your program must have.

- **30 pts - Fix errors from previous assignment:** The errors from the previous two assignments should be fixed. If your previous assignment was already perfect you do not need to do anything here to get full points.
- **30 pts – Texture Generation:** Download and read the paper:  
[http://www.cc.gatech.edu/~turk/my\\_papers/reaction\\_diffusion.pdf](http://www.cc.gatech.edu/~turk/my_papers/reaction_diffusion.pdf)  
 Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion  
 This paper has two parts, the generation of textures in the plane and the generation of textures on surfaces. You should implement the first part, the generation of textures in the plane. This program can be independent of the first two projects.
- **Parameter Control:** You should identify which parameters are important to control the pattern generation and make it possible to determine / adjust the parameters for the pattern formation interactively.
- **Four selectable parameter settings.** Besides setting parameters for the pattern formation interactively, you should also allow the user to select from at least four pre-specified different parameter sets. These four parameter sets should lead to four different patterns that should be similar or identical to the ones shown in figures two and three of the paper.
- You should provide sufficient user control so that you visualize how these pattern are formed.
  - There should be one mode where you start pattern creation and just show the results

- There should be one mode where you show each  $n^{\text{th}}$  time step of the pattern formation. The parameter  $n$  should be selected with a pop-up menu.
- There should be one mode where you show one new step every time the user hits the space key.