Advanced Graphics

SER 431 Fall 2018

1. Project Guidelines

- Have fun, learn, be creative, and create something you are proud to show-off!
- You will work in teams of 3. Define a product owner, a manager for developing and a manager for testing.
- Testing is important. Your project should work not only in the computer that is being developed but, in the TA, or instructor computer and in any lab computer. Do an extensive testing.
- It is ok to discuss the projects with other students. You can use the discussion board on 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 could be asked to demo any of your projects. In a 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.
- You will upload your projects on myASU. Details about this will be announced in class.
- Do not miss due dates! There is not make up projects or extensions. The myASU timestamp will show when you uploaded your project.
- 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, etc. Review the syllabus section regarding ASU policies in this matter.

2. Getting Started

- The projects assume that you are using: Microsoft Windows, Visual C++, and OpenGL (glut32.lib). Review SER 332 for details about how to use and install glut32 library.

  http://javiergs.com/teaching/ser332/

- To test your installation a first good step is to download and install two sample programs from SER 332 GitHub repository.

3. Programming: C and C++

- If you take SER 332, you are already familiar with C/C++. If not, you should learn it by yourself. 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.

4. Upload your projects

- Do not send the program as email attachment to the instructor or TA!
- Your project should be packed in a zip file
- The zip file should include all the source code and should be ready to compile.
- 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, among others, the following: (1) special functions that you implemented; (2) changes from the original specification that you negotiated with the instructor or TA to the original specification; (3) other things you consider important; (4) 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.

5. Game Requirements

- Read "What makes a game good?" – http://www.thegamesjournal.com/articles/WhatMakesaGame.shtml
  Apply tall hese ideas to your game

- Review ideas from diverse sources. Get inspired! For instance: https://www.youtube.com/watch?v=aviL3HX3UEc

6. Core Requirements (Part 1)

6.1. Graphics

- Basic computer graphics concepts and a first interactive computer graphics application. Design a game in which you can control one moving object, e.g. a *car*. You are free to change the interpretation of the project and write a similar application, e.g. a space ship

flying through space, a person walking through a city, a person walking through a garden, etc. **Part 1 is is the basis for the Part 2 and so on.**

- The scene should be modeled in three-dimensions.

- Your window should be split in two: a control panel view showing score or game related controls and a main view showing the game. **Game related controls are not the same that Graphics configuration options (described below).**

- 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.

- Define a menu (it could be pop-up menu or menu bar, or something else)

- Define keyboard controls and mouse controls to control the application. See below for details.

6.2. Functionality

- 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".

- **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.

- **Menu:** Implement a pop-up menu to control the application. The content of the pop-up menu is described in the following.

- **Two Viewports:** - Create two viewports. One for the game 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.).

- **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 normals and texture coordinates) have to be stored as vectors. **Unless other instruction is explicitly given.**

- **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. You will need vertex positions, vertex colors, per face normals, per vertex normals, and per vertex texture coordinates.

- **Create a Game:** The Game 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. Review your SER332 notes. That means you write a respective functions that creates a mesh data structure describing the object (e.g. createBox(...)).

- The Game needs a *Start Area* (e.g. color box on the ground) and a *Finish Area*. You could begin by placing boxes and cylinders to form walls in the game. Later you could replace them with more complex mashes. **Do not start with complex shapes, keep the simple ones (boxes, cylinders, and plains at hand). Simple shapes will help you to do testing.**

- **Object Loader:** Use or implement a file loader to load several unique 3D objects that will appear in the Game. 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. You can load the car as 3D mesh taken from the Internet or use a supplied model on blackboard if you want.

- **Main Object 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 menu with the choices of *quit* or a pick speed submenu. In this submenu, allow for slow, medium, and fast.

- **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.

- **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.

- **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.

- **Timer**: Implement a timer. **For instance**, 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.

- **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 game in a double orbit, e.g. an insect rotating around a dog which moves around a mailbox.

- **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.

- **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.

7. Core Requirements (Part 2)

7.1.Graphics

- Add more complex visual appearance to the program implemented in Part 1. That includes three-dimensional navigation, Normal calculation, Lighting, and Textures.

- For shading and lighting you will need: (1) a light source, (2) to specify materials for all geometry in the scene, and (3) (normalized) normals for all geometry in the scene.

7.2.Functionality

- Here are the elements that your program must have. Despite this "laundry list" of requirements, 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".

- **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 **Part 1** to include texture coordinate generation.

- **Texturing**: Apply textures to the scene. A texture is loaded from a **file** or created with a **procedure**. 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: **(1)** in the first version the color of the texture defines the final color of a pixel (of the polygon). ("Texture - Replace"); **(2)** in the second version the color of the texture is modulated (multiplied) with the color computed by the lighting and shading equations. ("Texture - Blend")

- **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.

- **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.

- **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.

- **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 then the crease angle. The models that you use also have to have some sharper edges so that this calculation makes sense.

- **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.

- 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.

- **Light source navigation mode:**
  If the user selects "Navigation - Light Source" the use 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).

- **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.

- **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).

8. Core Requirements (Part 3)

8.1.Graphics

- Your final submission should fix errors from previous the previous two parts to get full points.

8.2.Functionality

- **Texture Generation:** Download and read the paper:
  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 patterns are formed.

9. Advanced Requirements

This document describes the basic (core) requirements that you are able to start implementing right now (first week of classes). Advanced requirements will be added to each of the 3 releases. They will be explained in class. **The following are just examples of what these "advance" requirements will look like**:

- **Oriented Bounding Box Computation.** 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.

- **Collision Detection.** A collision detection algorithm detects collisions between your car and any other objects in the Game (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.

- **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.

- **Procedural Ground Plane Generation** 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.

As you can see, the core elements are needed in order to be able to work with advance requirements. Start working with your teams as soon as you can. Do not hesitate to ask questions and clarify doubts.

10. Grading

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.)