3D Software Pipeline Project

Topics covered

- 3D transforms
- Frustum plane equations
- Spherical bounding volume construction and transformation
- Culling models using spherical bounding volume
- Clipping triangles against frustum plane equations
- Depthbuffer algorithm
- Back-face removal
- Rasterization of triangle and line primitives

Software Framework

- A codebase that abstracts a software-based graphics pipe is provided for this assignment. The tasks described in the project specs below are completely implemented by the framework. You can implement the project specs in any order. Thus, while completing a certain task, you can use the framework's implementations for other unfinished tasks to test your current task.
- The software implementation of the graphics pipe renderer is included as library file `GfxLib.lib` [in directory `$(SolutionDir)$(Configuration)`]. The project will create an executable program by linking the software emulator library to the object code generated by your contribution in `GfxDriver`.
- The framework is completely based on the OpenGL based graphics pipeline discussed in class lectures. Just as with GLFW, `GfxLib` will create color and depth buffers and the store for texture images. All coordinate systems described in class lectures will work as is – there are no surprises except for one [see below].
- OpenGL has always used a different coordinate system than computer hardware, display devices, and Windows to reference and index two-dimensional entities such as windows, viewports, color and depth buffers, and textures. Windows uses the upper-left corner as origin with $x-$axis oriented right and $y-$axis oriented *down*. Recall that OpenGL describes its viewport and texture coordinates using the lower-left corner as origin with $x-$axis oriented right and $y-$axis oriented up. This means that a pixel [or a texel] (x, y) generated by OpenGL [and the framework] cannot be rendered as is. Instead, the pixel's y coordinate must be mapped to the coordinate system used by Windows. You must perform this mapping before accessing the color buffer, depth buffer, and a texture image.
- Since this framework was written before GLM was invented, it uses homegrown `Vector3`, `Vector4`, and `Matrix4` classes to implement the mathematical elements of a graphics rendering pipeline. GLM has now been integrated into the project - you can use GLM functionality in your code simply by including the appropriate headers in source files without modifying the project's properties.

What is to be implemented?

- *Triangle rasterizer* with the following features: back-face culling, depth buffering, color, and texture coordinate interpolation. Assume *repeat* wrap mode for texture coordinates along both s and t axes. Assume vertex color coordinates (r, g, b, a) are in range $[0, 1]$. You will have to implement ~~two~~ one interpolation technique~~s~~: *linear* barycentric interpolation [equivalent to GLSL `noperspective` interpolation qualifier and equivalent to your previous assignments] ~~and *hyperbolic* barycentric interpolation (equivalent to GLSL `smooth` interpolation qualifier)~~.

- *Wireframe rasterizer* that uses Bresenham algorithm to render edges of triangles.
- *Spherical bounding volume construction and transform*. Use Ritter's method to compute model bounding spheres. Once a bounding sphere is constructed, it must be transformed into the destination frame in which the object is culled. Assume affine transforms on objects are always applied in the following order: scale, followed by rotation, followed by translation.
- *Six frustum plane equations* of the frustum in view frame are computed using only the matrix manifestation of perspective (or, orthographic) transformation.
- *View-frustum object culling* using spherical bounding volumes in view frame.
- *Convex polygon clipping* using Sutherland-Hodgeman algorithm.
- *First person camera*.
- ~~*Object picking* using the ray casting method. Note that picking cannot be tested without first implementing the function that constructs model bounding spheres.~~
- ~~*Generate and render planar shadows* using the methods discussed in class.~~
- A correct implementation of the specs is provided. The also sample implements picking and planar perspective shadows [you're not required to implemented these features].
- For a complete list of user controls, see the definition of functions `MyMouseHandler()` and `MyKeyHandler()` in `$(SolutionDir)/GfxDriver/src/main.cpp`.

Step-by-step implementation

The first two steps must be implemented while the remaining steps can be implemented in any order you choose.

- The VS 2022 solution `csd2100+proj.sln` contains a single project labeled `GfxDriver`. The project's properties [in both Debug and Release configurations] are setup to include a simplistic software implementation of OpenGL [in `$(SolutionDir)$(Configuration)/GfxLib.lib`] for both Debug and Release configurations; setup of geometric models and textures; Windowing and I/O facilities; and GLM. The software emulator's header files are accessible from the project and are located in `$(SolutionDir)GfxLib/gfx`. Open the solution, and then compile, link, and execute the project. You should see the scene displayed [although with a reduced framerate]. Unlike the sample, the default implementation will not be able to pick objects. To improve framerate and to get picking to work, you'll have to implement auxiliary functions `gfxModel::ComputeModelBVSphere()` and `gfxSphere::Transform()` [in the next step].

  > ***The software pipe requires spherical bounding volumes to implement view frustum culling, clipping, and picking. Therefore, you must first implement both auxiliary functions*** `gfxModel::ComputeModelBVSphere()` ***and*** `gfxSphere::Transform()` ***[described below] before proceeding to implement other functionalities required by the spec.***

- *Bounding spheres*: Complete the following steps in file `$(SolutionDir)/GfxDriver/src/YourBoundingSphere.cpp`:
  - `gfxModel::ComputeModelBVSphere()`: This function is called once for each model at program startup to enclose the model's position coordinates in a bounding sphere constructed using Ritter's method. Note: Computing the bounding sphere using a trivial method involving AABBs will result in significant deductions.

  - `gfxSphere::Transform()`: This function transforms the model's bounding sphere with the corresponding object's geometric transform [recall that an object is an instance of a model] into a reference destination frame. Assume the sequence of affine transforms that may be applied to the model's instance will always consist of scale [if any], followed by rotation [if any], followed by translation [if any].
- *Culling*: Before transferring an object's geometry and render data to the graphics pipe, the bounding volume must be checked for trivial acceptance or trivial rejection with respect to the frustum. Trivially rejected objects are discarded while trivially accepted objects are not clipped. Objects that are neither trivially accepted nor rejected will intersect with the frustum and must therefore be clipped. To facilitate trivial acceptance or rejection tests, implement the following functions in file `$(SolutionDir)/GfxDriver/src/YourCullerClipper.cpp`:
  - `ComputeFrustum()`: This function computes the six plane equations of the frustum in view frame.
  - `Cull()`: Using an object's bounding sphere and the graphics pipe's frustum, this function first computes the bounding sphere's *outcode*. Using this *outcode*, the function will determine an object's status with respect to clipping: *trivial acceptance*, or *trivial rejection*, or *intersection*.
- *Triangle clipping*: If an object is neither trivially accepted nor rejected [as reported by previous function `Cull()`, the object's triangles must be clipped. Implement the following functions in file `$(SolutionDir)/GfxDriver/src/YourCullerClipper.cpp`:
  - `Clip()`: Clips object triangles using Sutherland-Hodgeman algorithm. ~~It is required that your implementation of `Clip()` use the `ClipEdge()` function.~~
  - To test your culling and clipping implementation and for correct submission, uncomment the following line in `$(SolutionDir)/GfxDriver/src/main.cpp`:

    ```
    //#define YOUR_CULLER_CLIPPER
    ```

- *Triangle and wireframe rasterizer*: Implement a triangle and wireframe rasterizer in file `$(SolutionDir)/GfxDriver/src/YourRasterizer.cpp` with the following features:
  - Back-face culling, depth buffering, texture coordinate interpolation, vertex color interpolation, and texture wrapping.
  - Vertex color and texture coordinates must be interpolated using ~~both~~ linear ~~and hyperbolic~~ barycentric interpolation method.
  - Some objects in the scene are texture mapped while others are lit. There are no objects that are both lit and texture mapped. The stub function `DrawFilled()` provides details on how to determine if the triangle is texture mapped or lit.
  - In wireframe mode, use Bresenham algorithm to complete definition of stub function `DrawLine()`. The function must render pixels using black color.
  - To test your rasterizer, uncomment the following line in `$(SolutionDir)/GfxDriver/src/main.cpp`:

    ```
    //#define YOUR_RASTERIZER
    ```

- *First person camera*: The camera described in class lectures is suitable for applications such as flight simulators. For this project, you will instead implement a slightly different camera. This camera should allow a character to walk on a planar floor even if the character is looking up into the sky. This behavior is different than a flight simulator where the aircraft is displaced in the direction in which it looks.

- Read declaration of base camera class, `gfxCamera`, in `$(SolutionDir)/GfxLib/gfx/Camera.h`.
- Read declaration of derived camera class, `YourCamera`, in `$(SolutionDir)/GfxDriver/src/YourCamera.h`.
- Implement the derived camera class, `YourCamera`, in `$(SolutionDir)/GfxDriver/src/YourCamera.cpp`.
- To test your implementation [as opposed to the framework implementation], uncomment the following line in `$(SolutionDir)/GfxDriver/src/main.cpp`:

```
//#define YOUR_CAMERA
```

- ~~*Planar shadows*: Generate planar perspective shadow geometry by completing the following steps in file `$(SolutionDir)/GfxDriver/src/YourPlanarShadow.cpp`:~~
  - ~~`GetShadowGeometry()`: This function is called during the transformation stage and it transforms model frame occluder position coordinates to clip frame. First, the receiver is rendered normally with depth buffer on, then shadows are enabled in the graphics pipe (in `main.cpp`) and the receiver is rendered a second time with depth buffer off. When shadows are enabled, the transformer short-circuits its own computations and calls `GetShadowGeometry()` to obtain clip frame position coordinates for the occluder. You will need to compute a 4×4 matrix manifestation of the perspective shadow transform using the view frame shadow receiver plane equation and view frame point light source position. To test your planar shadow implementation, uncomment the following line in `$(SolutionDir)/GfxDriver/src/main.cpp`:~~

```
//#define YOUR_PLANARSHADOW
```

- ~~*Picking*: To incorporate 3D picking functionality into the project, complete the following steps in file `$(SolutionDir)/GfxDriver/src/YourPicker.cpp`:~~
  - ~~`Pick()`: This function takes as arguments a pick point (or pixel) (x, y), a list of scene objects, and a pointer to camera. You must implement the ray casting method discussed in class lectures to return the picked object's index from the list of scene objects, or -1 if no object is picked. For this submission, choose the smallest positive intersection time between pick ray and bounding spheres of objects. See the function header for additional details. To test our picking implementation, uncomment the following line in `$(SolutionDir)/GfxDriver/src/main.cpp`:~~

```
//#define YOUR_PICKER
```

Submission

1. You *must* submit the following source files: `YourCamera.h`, `YourCamera.cpp`, `YourBoundingSphere.cpp`, `YourCullerClipper.cpp`, and `YourRasterizer.cpp`. If you've modified the associated header files, don't forget to include them too.
2. The files must be placed in a directory labeled as: **<student-login-name>-<course>-<final>**. If your DIT student login is **foo**, then the directory would be labeled as **foo-csd2100-final**. *Zip* the folder and upload it to the submission page on the course web page.

Grading Rubrics

Competency **core1** simply says that if your source code doesn't compile nor link nor execute, there is nothing to grade. Competency **core2** indicates whether you're striving to be a professional software engineer, that is, are you're implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [file and function headers properly annotated]. The remaining rubrics are related to the tasks assigned in this assignment.

- [**core1**] Submitted files must build an executable file with ***zero*** errors. This rubric is not satisfied if the generated executable crashes or displays nothing.
- [**core2**] Submitted source code must satisfy ***all*** requirements listed below. Any missing requirement will decrease your grade by one letter grade.
  - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
  - Source code file submitted is correctly named.
  - Source code file is *reasonably* structured into functions and *reasonably* commented. See next two points for more details.
  - If you've created a new source code file, ***it must have file and function header documentation blocks.***
  - If you've edited a source code file provided by the instructor or from a previous assignment, the file header must be annotated to indicate your co-authorship and the changes made to the original or previous document. Similarly, if you're amending a previously defined function, you must annotate the function header to document your amendments. You must add a function header to every new function that you've defined.
- [**core3**] Bounding spheres are correctly computed using Ritter's algorithm. ***Implementation using a trivial algorithm involving AABB or similar scheme will not be accepted.***
- **[core4]** Wireframe images must be rendered using Bresenham algorithm with pixels rendered in black color.
- [**core5**] Triangle rasterization [with depth buffering, back-face removal, vertex color and texture coordinates interpolated using barycentric interpolation method, and repeat texture wrap mode] is correctly implemented. Rasterization artifacts such as holes and dropouts will be significantly penalized.
- [**core6**] Object culling is correctly implemented. Frustum plane equations should be computed as described in class lectures. Computing plane equations using naïve methods will result in significant deductions. Bounding spheres must be correctly transformed into a destination reference frame. Incorrect transforms of bounding spheres will result in significant deductions.
- [**core7**] Triangle clipping is correctly implemented using Sutherland-Hodgeman algorithm. Clipping artifacts such as tears and holes will be significantly penalized.

Mapping of Grading Rubrics to Letter Grades

The following table illustrates the mapping of core competencies listed in the grading rubrics to letter grades:

Grading Rubric Assessment	**Letter Grade**
There is no submission.	F
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If **core2** rubrics are not satisfied, final letter grade will be decreased by one. This means that if you had received a grade A and **core2** is not satisfied, your grade will be recorded as B, an $A-$ would be recorded as $B-$, and so on.	
If **core3** is not implemented, then the main learning outcomes of this assignment cannot be assessed.	F
If none of the four rubrics [**core4** thro' **core6**] are correctly implemented.	F
One of three rubrics [**core4** thro' **core6**] is correctly implemented.	D
Two of three rubrics [**core4** thro' **core6**] are correctly implemented.	C
Three of three rubrics [**core4** thro' **core6**] are correctly implemented.	B
core7 rubric is correctly implemented.	$A+$
Note that your implementation will be tested with different scenes. You can modify the scene [by adding additional objects for instance] by amending the code in `InitScene()` defined in `$(SolutionDir)GfxDriver/src/main.cpp`.	