MTAT.03.015 Computer Graphics (Fall 2013) Exercise session V: Projection & Vertex Shaders

Konstantin Tretyakov, Ilya Kuzovkin

October 7, 2013

Today we shall study the vertex transformation from world space to clip space in detail. First we focus on its final step – the projection transform, and then look at how the complete transformation can be specified using vertex shaders. As usual, the base code is provided in the `practice05.zip` archive on the course website or via the course github page. Download, unpack and open it. You will need to submit your solution as a zipped archive file.

1 Clip Volume Normalization

Projection is the last step of the vertex transformation (*vertex shading*) pipeline. In classical OpenGL it can be specified using a 4×4 *projection matrix* $\mathbf{P}$, that is applied to vertices right after the model-view matrix $\mathbf{VM}$. That is, given the vertex with original homogeneous coordinate vector $\mathbf{x}$, it is transformed to *normalized device coordinates* $\mathbf{x}_{\text{nd}}$ as follows:

$$\mathbf{x}_{\text{nd}} = \mathbf{PVMx}\,.$$

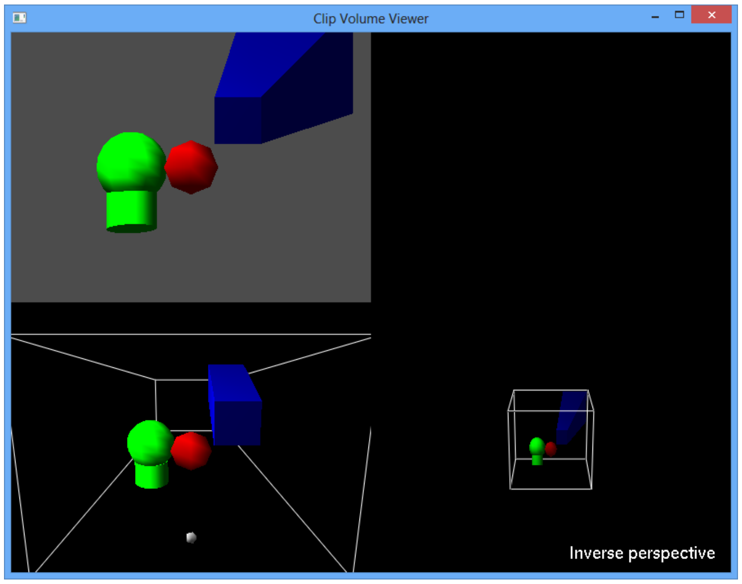
After this transformation, everything that falls into the cube $[-1,1]\times[-1,1]\times[-1,1]$ is rendered onto the screen using standard 2D rasterization methods.

It is convenient to visualize the projection transformation in terms of the *clip volume* that it imposes on the scene. That is, the original volume that will be mapped to the $[-1,1]\times[-1,1]\times[-1,1]$ cube after projection.

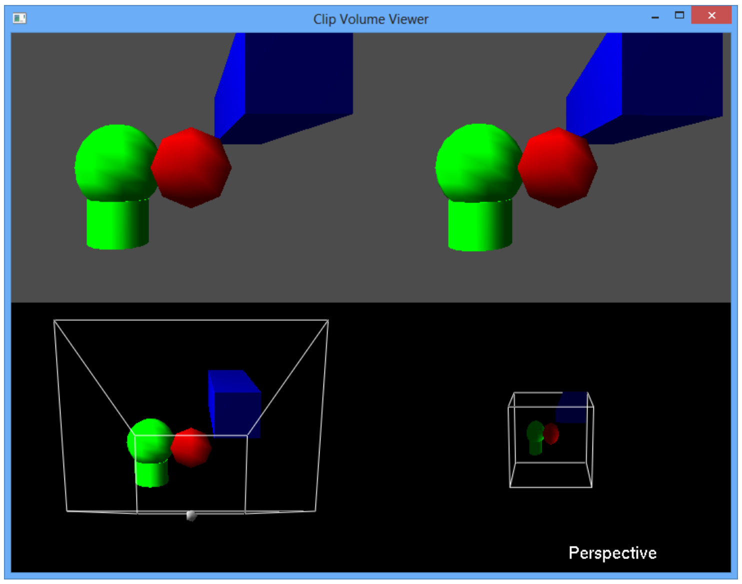
To understand this concept, open and run the project `1_ClipVolume`. The application shows a simple scene rendered on the top left, the three dimensional view of the clip volume from a distance (bottom left) and the view of the normalized clip space after projection (bottom right). Use the spacebar to change between different projections. Examine the shape of the clip volume before projection, after projection, and see what happens to the vertices within the clip volume during projection.

Exercise 1 (0.5pt). The three different projections are described in the functions `scene_<xxx>_projection`, using OpenGL matrix multiplication routines. Currently, the function `scene_perspective_projection` uses a call to `gluPerspective` to specify the view frustum. Replace this with a call to `glFrustum`, that would result in exactly the same clip volume.

Exercise 2* (0.5pt). Implement the function `scene_inverse_perspective_projection`. It should specify an *inverted* perspective. That is, the view frustum should be exactly like the one of the original perspective, but with the front wider than the back. As a result, the dimensions of objects on the rendered scene will be *greater* for objects further away from the camera. The picture you should aim for is shown below.



Exercise 3 (0.5pt). The top-right viewport of the screen is currently free. We shall use it to implement a *cross-eyed*[1] 3D view of the scene. Render into that viewport the same scene as is rendered in the top-right viewport, but with the camera rotated slightly to the left around the center of the scene (which is $(0, 0, -5)$ in camera coordinates). The result should look as shown below.



Hint: If the use of `glViewport` and `glScissor` is not clear from the already existing code, do not hesitate to read the docs.

Exercise 4* (1pt). You should notice that the objects are colored differently in the lower right window: for perspective and inverse perspective projections the objects seem to be darker than in all other renderings. Explain why it is the case.

Hint: To solve this task you need to know that the colors of the vertices are determined by their position relative to the light source. The light source position is specified by the `glLightfv` line in the code (you'll have to find it and understand what is happening). Try changing the light position to be behind the camera and see what happens.

[1] `http://www.starosta.com/3dshowcase/ihelp.html`

2 Vertex Shading and GLSL

The procedure of vertex transformation into clip space is known as *vertex shading*, and can be customized in all modern graphics cards. The customization is done by providing a *shader program*, that has to be written in a special C-like language – *GLSL*, the *GL Shading Language*.

Open the project `2_HelloGLSL` and study its structure. Although it should remind you the simplest GLUT "Triangle" program from the last week, there are three important changes.

1. Firstly, the project uses the GLEW library in addition to GLUT. *GLEW (GL Extension Wrangler)* is necessary to enable the cross-platform use of most OpenGL features beyond the most basic ones. GLEW is enabled in a program by doing three simple things:

   - Writing `#include <GL/glew.h>` in your program.
   - Invoking `glewInit()` during initialization.
   - Adding `glew32` to the list of libraries to link with (in the *Build Options* dialog).

   Once it is done, you can automatically use all of the newer OpenGL functions, supported by your graphics card.

2. Secondly, the project consists of more than one source file. Besides `hello_glsl.cpp`, which contains the main code of the program, there is `shader_util.cpp`. This file has some useful functions and classes, which you do not need to worry about. Note, however, that the *declarations* of those functions and classes are given in a separate *header file*, `shader_util.h`. This file is `#include`-d in the main program, which lets you use the functions, just like you do with system libraries.

3. Finally, and most importantly, there is a separate file, `simple_vertex_shader.glsl`, which contains the code for the custom vertex shader program in GLSL. This shader is loaded and enabled in the `display` function using the code:

   ```
   shader_prog shader("../src/simple_vertex_shader.glsl", NULL);
   shader.use();
   ```

GLSL Basics

Study the code in `simple_vertex_shader.glsl`. You will see something, that resembles a tiny C program. This is GLSL, a limited version of C. The shader operation must be specified in the function `main`. Within this function you may use basic arithmetics, declare variables, use `if`, `for` and `while` constructs, etc, just like in C. Besides the basic types (`int`, `float`, etc) there is a number of built-in data structures and functions for working with vectors and matrices. A nice description of them is given here: `http://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations`.

The aim of a vertex shader is to compute the transformed position of a single vertex. The input vertex is provided in the built-in *input* variable[2] `gl_Vertex`. The transformed value must be saved into the built-in *output* variable `gl_Position`. The built-in function `ftransform()` implements the default fixed-function model-view-projection transformation.

A number of other useful variables are available[3]. The variable `gl_ModelViewProjectionMatrix`, for example, is the model-view-projection matrix that you constructed in your OpenGL code.

Besides built-in variables, you can provide your own data to the shader from the program. In general, there are differences in how it is done depending on the kind of data[4], but in this practice we shall only study a simple example of a *uniform* variable (you are suggested to read the manual about *in* variables too, though).

A *uniform* variable is a variable that is common to all the vertices. You first need to declare it in the shader using the `uniform` keyword (check out the third line in the GLSL file). Then you have to bind the data from your program to this variable using functions `getUniformLocation` and `setUniform<xxx>` (check out the commented out code in the `display` function).

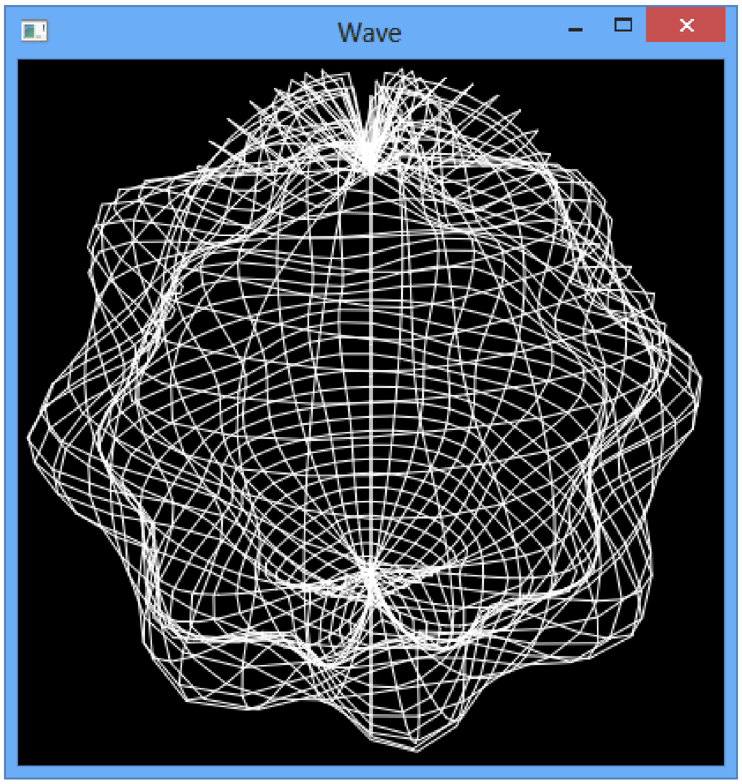
Exercise 5 (1pt). Play with the code provided in `simple_vertex_shader.glsl`. Try uncommenting the different possibilities and see how they affect the result. To get the points for the task, change the shader so that the vertices would follow some funny trajectory, in the spirit of the last example shown there. For example, make them crawl along the edges of the screen. You are free to introduce custom *uniform-* or *in-* variables to the shader, if you want.

Exercise 6 (1pt). Open the project `3_Wave`. It currently renders a simple wireframe sphere. Your task is to design a vertex shader (in the file `wave_vertex_shader.glsl`) that would make the sphere "wavy". You are free to interpret the concept of "waviness" to your liking, but the kind of thing you should aim to get is illustrated on the picture below:

[2] `http://www.opengl.org/wiki/GLSL_Predefined_Variables_By_Version#Vertex_shader_attributes`

[3] Later versions deprecated most of them, though, in favor of manual variable binding.

[4] `http://www.opengl.org/wiki/Type_Qualifier_(GLSL)`



You are welcome to have the wave actually oscillate (for this purpose there is already a `time` uniform set up in the shader for you).

Hint: What you want to do is to shift the position of each vertex back and forth according to a sinewave along the direction of the normal at this vertex. The normal direction can be obtained using the `gl_Normal` built-in variable. The phase of the sinewave could depend on the z coordinate and/or the `atan(y, x)` angle. You can also try multiplying sinewaves together to achieve the necessary effect.

Hint: It seems that the `gl_Normal` vectors that FreeGLUT outputs for the vertices of the sphere do not have their fourth element equal to zero (looks like a FreeGLUT bug to me). This may result in unexpected results. To fix this do something like

```
vec4 normal = vec4(gl_Normal.xyz, 0);
```

Note that if you develop a complex scene with several objects, it is completely normal to have a separate shader for each object. For example, you could have several wavy spheres flying around, each with its own "waviness" pattern.