Getting Started with Assignment 2c

- The provided template includes a 3D model, and code to:
  - read that model in from a file;
  - scale the model to fit in a [–1,1] cube
  - render the model using the identity matrix for the view and projection matrices
  - shade each fragment using a very simple diffuse lighting model, so that the geometry can be better understood than with flat shading.

Getting Started with Assignment 2c

- What you are asked to do is:
  1. Determine an appropriate initial viewing position and orientation for a first-person walkthrough, based on the understanding that 1 unit = 1m in the model and that the origin is currently in the center of the model.
  2. Dynamically define a current viewing transformation matrix using the eye position, view direction, and (0, 1, 0) as up direction
  3. Define a suitable perspective projection matrix

Getting Started with Assignment 2c

- What you need to do (continued):
  4. Apply the viewing and perspective transforms in the vertex shader
  5. Allow the user to interactively modify the eye position and view direction using the arrow keys
  6. Maintain an undistorted view of the scene if the user changes the window size

Getting Started with Assignment 2c

- Since we are now working with a model that is defined in 3D, you will need to perform hidden surface removal so that closer surfaces appropriately occlude farther ones.

- The following extensions of hw2b are required for this:
  - enable depth testing (typically done in the `init()` function):
    ```
    glEnable(GL_DEPTH_TEST);
    ```
  - clear the depth buffer when re-drawing the display:
    ```
    glClear( ... | GL_DEPTH_BUFFER_BIT);
    ```

Getting Started with Assignment 2c

- The template already includes these extensions, so at this point, without doing anything fundamentally new, you should be able compile and use the template to render a 3D model.

  [To think about: How can this appear to work, even though you have not explicitly defined a projection matrix? When the identity matrix is considered to be the projection matrix, what happens to the depth values?]

Getting Started with Assignment 2c

- To allow the user to interactively control the eye position and view direction, you will need to:
  - maintain an appropriate viewing transformation matrix in your application program, and
  - send the updated viewing transformation matrix to the vertex shader, where you will use it to transform the vertices in your scene from world coordinates to camera coordinates

Getting Started with Assignment 2c

- For simplicity, you are allowed to use global variables to manage the camera location and the viewing direction. Appropriate initial viewing parameters may be defined in advance and hardcoded or read from a file.
- Even though in this program I'm not asking you to implement the ability for the user to look up or down, for general extensibility you are advised to structure your code in a way that could be accommodating to such view direction changes.

Getting Started with Assignment 2c

- In your `init()` procedure, you should initialize the viewing parameters and set up the viewing transformation matrix ***V***

$$V = \begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $n = –normalized\_view\_direction$,
$u = up \times n$ (normalized), $v = n \times u$, and
$d_x = –(eye \cdot u)$, $d_y = –(eye \cdot v)$, $d_z = –(eye \cdot n)$.
Remember that *u*, *v* and *n* must all be unit vectors.

Getting Started with Assignment 2c

- You will want to pass the matrix ***V*** to the vertex shader as a uniform variable, and use it in the vertex shader to transform the vertex locations.
- You can do this using exactly the same syntax that you used in assignment 2b for the model matrix ***M***.

Getting Started with Assignment 2c

- You need to define the matrix variable location for use in the shader:

```
v_location = glGetUniformLocation( program, "V");
```

- and, you have to transfer the values in the matrix to the GPU:

```
glUniformMatrix4fv(v_location, 1, GL_FALSE, V);
```

(note that I am assuming above that the viewing matrix is defined using the variable name V. You can of course use whatever name you see fit.)

Getting Started with Assignment 2c

- Inside your vertex shader, you will then need to define the 4×4 matrix $\boldsymbol{V}$ as a uniform variable:

```
uniform mat4 V;
```

- and, you will need to multiply the vertex positions by it. Although this assignment doesn't require you to use any model transformations, if you did want to include them, you could:

```
gl_Position = V*M*vertex_position;
```

(note that the model transformations would need to be applied first)

Getting Started with Assignment 2c

- The next step in implementing hw2c is to enable the user to move the camera location forward and back in the viewing direction using the ‘up’ and ‘down’ arrow keys.
- You will need to set up a callback function to handle events that are generated when an arrow key is pressed, just as you did in assignment 2b.
- In this callback function, you will need to check which key was pressed, and then modify the eye position accordingly.
- Essentially, you can just replace the code you used to enable scaling in hw2b with new code that modifies the view.

Getting Started with Assignment 2c

- Note that the viewing transformation matrix ***V*** depends on the location of the eye; this means that when the eye position changes, you will need to update ***V***.

$$V = \begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$d_x = -(eye \cdot u)$$
$$d_y = -(eye \cdot v)$$
$$d_z = -(eye \cdot n)$$

Getting Started with Assignment 2c

- Next, you will want to allow the user to rotate the viewing direction to the left and right using the ‘left’ and ‘right’ arrow keys.
- Since the viewing direction starts at the camera origin, you can re-define the viewing direction by using a simple rotation of the viewing direction vector around the *y* axis, to obtain a new viewing direction vector.

$$\begin{bmatrix} new\_vdir_x \\ new\_vdir_y \\ new\_vdir_z \\ 0 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} viewdir_x \\ viewdir_y \\ viewdir_z \\ 0 \end{bmatrix}$$

Getting Started with Assignment 2c

- Once the viewing direction has been changed, you will need to update the viewing transformation matrix $\boldsymbol{V}$. Note that the u, v, and n axes will all need to be re-defined, as well as d, which depends on u, v and n.

$$V = \begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$n = -normalized\_view\_direction$

$u = up \times n$ (normalized)

$v = n \times u,$

$d_x = -(eye \cdot u), d_y = -(eye \cdot v), d_z = -(eye \cdot n).$

Getting Started with Assignment 2c

- The next step is to define and apply a perspective projection transformation.
- In your `init()` procedure, you can use initial values of the parameters *left*, *right*, *bottom*, *top*, *near*, *far* to define a perspective viewing frustum and set up the projection-normalization transformation matrix

$$P = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Getting Started with Assignment 2c

- Once you have defined the viewing transformation matrix ***V*** and the projection transformation matrix ***P***, you will need to apply them in your vertex shader.
- Note that after the perspective transformation, the value of *w* for each vertex position will no longer be guaranteed to be 1.0, so you will need to explicitly perform the perspective divide operation:

```
gl_Position =
  P*V*M*vertex_position/vertex_position.w;
```

Getting Started with Assignment 2c

- The final step is to correctly handle changes to the window size.
- The viewport defines the portion of the display window that will be filled by the rendered image. By default, the viewport is initialized to be the same size as the window.
- However, if the window size is changed by the user, the application program needs to update the viewport size to match the new window size in order to avoid distorting the view.

Getting Started with Assignment 2c

- One could also potentially consider the locations of the edges of the window on the screen when defining the edges of the viewing frustum to cause the window to behave as if it were a portal through which one can view a virtual scene
- But that is an extension for another time.

Getting Started with Assignment 2c

- By default, if the viewport size is left unchanged when the window size is *decreased*, portions of the original view will no longer fit within the window.

- By default, if the viewport size is left unchanged when the window size is *increased*, the newly exposed portions of the window may either be filled with a background color or with colors “left over” from prior content.

- If you want to either show more of the rendered scene within a larger window, or to view the same scene with a greater magnification in a larger window, you will need to explicitly enable this by adapting the viewport size.

Getting Started with Assignment 2c

- Please note that if your scene consists of a single object suspended in an empty void, the consequences of neglecting to update the viewport after increasing the window size may be masked.
- Therefore, you are advised to verify the correctness of your handling of window + viewport size changes from a point of view inside the cathedral model.

Getting Started with Assignment 2c

- Note that it will not be sufficient to only update the viewport size to match the window size when the window is reshaped; you must also re-define the viewing frustum used to render the viewport contents.
- Matching the viewport size to a new window size causes the already-rendered image to be re-scaled to fill the entire window exactly.
- This means that if the aspect ratio changes, the rendered image will be <u>distorted</u> to fit the new viewport/window.

Getting Started with Assignment 2c

- You can avoid stretching or squashing in the rendered image by re-defining the aspect ratio of the viewing frustum to match the aspect ratio of the newly reshaped window.
- When the window is made uniformly bigger or smaller, it is possible to re-size the view without distortion, but when the shape of the window changes, the scene needs to be re-rendered using a differently-shaped frustum that can completely fill the new window/viewport bounds.

Getting Started with Assignment 2c

- The first step in this process is to define a callback function that gets triggered when the window is resized. You can use:

```
glfwSetWindowSizeCallback(window,
window_resize_callback);
```

Getting Started with Assignment 2c

- The resize callback function receives two integer parameters representing the width and height of the newly reshaped window.
- On some systems, a new resize event is triggered for every successive pixel of displacement in the window shape, as the corner of the window is being dragged around
- On other systems, the resize event may not be transmitted until the corner of the window is released. If this is your situation, don’t worry about it.

Getting Started with Assignment 2c

- The first thing you want to do in your resize callback function is to re-define the viewport to match the new window size.
- You can do this with the `glViewport()` command, which takes the parameters *x*, *y*, *width*, *height*. The point (*x*, *y*) specifies the origin of the viewport within the window. Typically we use (0, 0). The new values of the window's *width* and *height* will be passed in to the callback function:

```
void window_resize_callback(GLFWwindow*
window, int width, int height)
```

Getting Started with Assignment 2c

- The next thing you want to do in your resize callback function is to calculate the aspect ratio of the new window and use that to re-define the *left*, *right*, *bottom*, and/or *top* extents of the viewing frustum.
- You can define *aspect_ratio* = *width* / *height*
- You may need to cast width and/or height to floating point numbers before doing this division operation to avoid the possibility of integer division, which could give you an undesired outcome.

Getting Started with Assignment 2c

- Assume that we start out with a square window. If resizing causes the *aspect_ratio* to become less than 1, we would then have *height* > *width*, and this means that the window should now show more of the scene content in the vertical direction.

- We can accomplish this by modifying the viewing frustum by increasing its height while leaving its width unchanged from the original settings.

Getting Started with Assignment 2c

- Alternatively, if we start out with a square window and resizing causes the *aspect_ratio* to become greater than 1, we would then have *width* > *height*, and this means that the window should show more content horizontally.
- We can accomplish this by modifying the viewing frustum to increase its width while leaving its height unchanged from the original settings.
- Changing the frustum in this way will cause the view of the scene to shrink or magnify as the window becomes larger or smaller overall.
- Alternative methods of handling window resizing without distortion are of course also possible.

Getting Started with Assignment 2c

- As the last step, you will need to update the projection transformation matrix using the new frustum extents before redrawing the scene.

Getting Started with Assignment 2c

- Tips on defining a suitable viewing frustum:
  - Try to avoid using an excessively small value for *near*, as an unnecessarily short distance from the eye to the near clipping plane will be wasteful of depth precision
  - Pay attention to the fields of view being defined by the parameters *near*, *left*, *right*, *bottom*, and *top*. As the geometric field of view (used for rendering) diverges from the display field of view (which depends on the window size and the viewing distance), the scene can begin to appear overly distorted. Remember: the distance *near* provides the context within which the distances *left*, *right*, *bottom* and *top* are used to define the fields of view.

Getting Started with Assignment 2c

- Tips on defining a suitable viewing frustum:
  - In situations where the eye is relatively far from the scene content being viewed, a larger value of *near* can be tolerated.
  - To avoid problems with the near clipping plane running into objects in a first-person viewing situation in which the eye *must* be close to objects in the scene, you can let the value of *near* be small, relative to the distance from the camera to objects in the scene, but then you will need to define the values of *left, right, bottom* and *top to* be similarly small, so that the viewing frustum retains a reasonable shape.