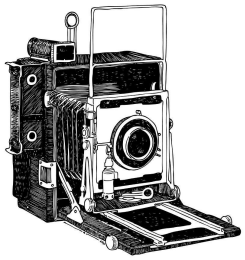
Lab 6 - Camtrans



Prologue

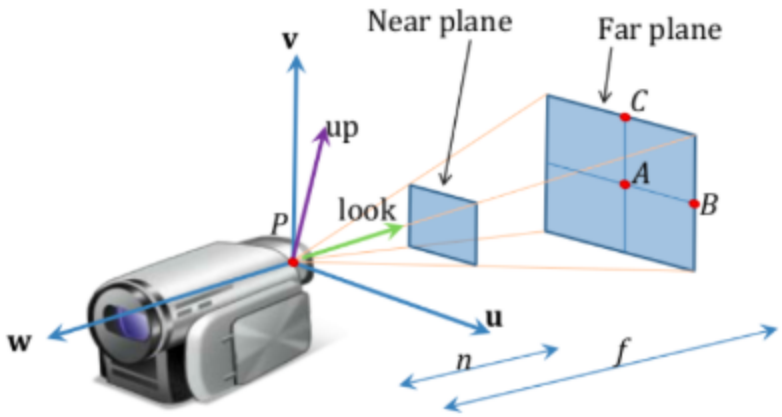
In this lab, you will create your own virtual camera. This camera will use various viewing properties to produce a transformation matrix to display 3D graphics on our 2D output devices. The code written in this lab will be used in future assignments.

Intro

When rendering a three-dimensional scene, you need to go through several stages. One of the first steps is to take the objects you have in the scene and break them into triangles. You did that in assignment 1. The next step is to place those triangles in their proper position in the scene. Not all objects will be at the "standard" object position, and you need some way of resizing, moving, and orienting them so that they are where they belong. There remains but one important step: to define how the triangulated objects in the three-dimensional scene are displayed on the two-dimensional screen. This is accomplished through the use of a camera transformation, a matrix that you apply to a point in three-dimensional space to find its projection on a two-dimensional plane.

The camera transformation typically also handles positioning and orienting the camera so that we can easily change where the scene is viewed from. It is entirely possible to leave out this component - all that you would have to do is make sure you position your objects such that they fall within the standard view volume. But this is tedious and inflexible. What happens if you create a scene, and decide that you want to look at it from a slightly different angle or position? You would have to go through and reposition everything to fit your generic camera transformation. Do this often enough and before long you would really wish you had decided to become a sailing instructor instead of coming to Brown and studying computer science.

For this assignment, you will be creating a camera object that provides the methods for several adjustments that one could perform on a camera. Once that has been completed, you will possess all the tools needed to handle displaying three-dimensional objects oriented in any way and viewed from any position.



Getting Started

This assignment will be completed within the code you have been using for your projects, mainly inside the /camera folder.

Since it is a part of the project code, you can run the demo using the regular project demo through FastX3 on department machines (`/course/cs123/bin/cs1230_demo`).

Demo

To see the Camtrans camera in action, switch to the 3D canvas, select the "Camtrans" tab, and make sure "Use orbit camera instead" is unchecked. The demo uses the following defaults for the camera's initial state.

- The near clip plane is 1 and the far clip plane is 30.
- The vertical view angle is 60 degrees.
- The screen aspect ratio is 1:1.
- The eye position of the camera is at (2,2,2) in world coordinates.
- The camera is looking at the origin, and its up vector is (0,1,0) in world coordinates.

The translate and rotate dials modify the camera's position relative to a virtual set of axes, which represent the camera's "local" coordinate space. In the camera's coordinate space, the camera is located at (0, 0, 0), the camera is looking along the negative Z axis, the Y axis is pointing upwards, and the X axis is pointing to the right. To emphasize the difference between the world's coordinate space and the camera's "local" coordinate space, the camera space axes are usually referred to as W, V , and U, respectively. When you move the camera, this set of axes moves with it.

- The "**<x, y or z>-axis**" buttons position the camera so that it is located two units along the **<x, y or z>** axis, pointing towards the origin.
- The "**Axonometric**" button positions the camera such that it is located at the point (2, 2, 2) and is pointing towards the origin.
- The "**FOV**" slider adjusts the camera's field of view by setting the height angle.
- The "**Near**" and "**Far**" sliders set the locations of the near and far clipping planes.
- Finally, the "**Aspect ratio**" label reflects the aspect ratio of the rendered image, which can be adjusted by resizing the window.

OpenGL Mathematics (GLM)

GLM is a C++ mathematics library which provides useful implementations of matrices and vectors as well as functions which can perform matrix transformations. In this lab, you should use GLM to store and transform matrices and vectors. Some of the classes/functions you can use are:

- `glm::mat4x4` - A 4x4 matrix.
  - `glm::mat4x4 mat = glm::mat4x4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);`
  - The default constructor for `glm::mat4x4` constructs an identity matrix.
- `glm::vec3` or `glm::vec4` - 3D and 4D vectors.
  - `glm::vec3 v = glm::vec3(1,0,0);`
- `glm::transpose(glm::mat4x4 m)` - Returns m^T, the transpose of matrix m.
- `glm::normalize(glm::vec3 v)` - Normalizes v and returns the resulting vector.
- `glm::dot(glm::vec3 v1, glm::vec3 v2)` - Returns the dot product of two vectors (v1 and v2).
- `glm::cross(glm::vec3 v1, glm::vec3 v2)` - Returns the cross product of two vectors (v1 and v2).
- `glm::radians(float degrees)` - Converts from degrees to radians.
- Trigonometry functions: `glm::sin(float rad)`, `glm::cos(...)`, `glm::tan(...)`, `glm::acos(...)`, `glm::asin(...)`, `glm::atan(...)`
- Overridden operators:
  - Adding vectors: `glm::vec3 result = v1 + v2;`
  - Scalar-vector multiplication: `glm::vec3 result = 2 * v;`
- You can access/mutate the components of a vector:
  - `glm::vec3 v = glm::vec3(2,3,4);`
  - `v.x // == 2`
  - `v.y // == 3`
  - `v.z // == 4`

However, you may not use the following GLM functions for this lab:

- `glm::lookAt(...)`

- `glm::perspective(...)`
- `glm::scale(...)`
- `glm::rotate(...)`
- `glm::translate(...)`

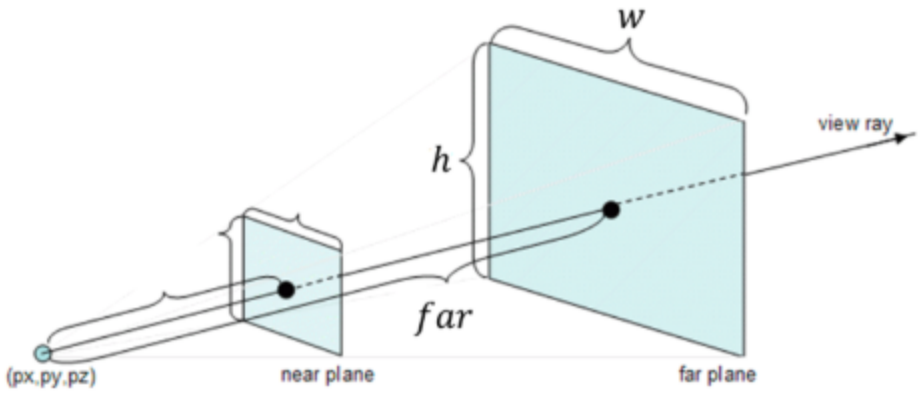
IMPORTANT: GLM and OpenGL use **column-major** matrix order. The lectures describe matrices in row-major order. This means that, if you write out matrices in row-major order when constructing GLM matrices, you should transpose them with `glm::transpose(...)` before storing them.

For example: `glm::mat4x4 columnMajorMatrix = glm::transpose(glm::mat4x4(/* row major matrix values */ ));`

Additional Notes / FAQ

Aspect Ratio

Below is the view frustum, where **p = (px, py, pz)** is the eye position, **near** is the distance from p to the near plane, and **far** is the distance from p to the far `plane`:



Here **w** and **h** are the width and height of the section of the frustum on the far plane, which is a distance **far** away from the eye position. The **aspect ratio** is the ratio of width to height:

$$\texttt{aspect} = w_{near}/h_{near} = w/h$$

To calculate the projection matrix, the lecture slides use the angles θ_w and θ_h, which are related to everything as follows:

$$w/2 = far * \tan(\theta_w/2)$$

$$h/2 = far * \tan(\theta_h/2)$$

Note that this means the aspect ratio is NOT equal to θw/θh. The aspect ratio relates the tangents of the angles, not the angles themselves. The GUI will give you the height angle θ_h in the `setHeightAngle()` method in */camera/CamtransCamera.h* . You can get θ_w by solving for it in the above equations, but you can also take a shortcut. Since the matrix from the lecture slides in Viewing III only uses tan($\theta_w/2$), you can solve for tan($\theta_w/2$) in terms of the aspect ratio and tan($\theta_h/2$) and avoid having to calculate θ_w at all.

Convert Degrees to Radians

Most (all) the angles we give you in methods like `setHeightAngle()` and `rotate<U,V,orW>()` are in degrees. You should convert these values to radians before using them. (math.h functions such as sin, cos, tan expect radians, as do glm functions because of a line in CS123.pro that will `#define GLM_FORCE_RADIANS`.)

- `glm::radians(degrees)` converts degrees to radians and returns the result.

Rotating the Camera

When you rotate the camera, you want to rotate the camera with respect to its own local coordinate system – not the world coordinate system; rotating the camera around any of its axes should not cause a translation in world space.

Memory Management

When you create a matrix like this: glm::mat4x4 m = glm::mat4x4();, you don't have to delete it because that entire matrix is stored on the stack. You only have to call delete when you call new yourself.

U, V, and W Vectors

The U, V, and W axes should always be perpendicular! In addition, the U, V, and W axes are not necessarily the same as your look, up, and right vectors. Make sure that when your camera is moved, you also re-adjust the look, up, and right vectors.

Camera

Task 1:

- In the CamtransCamera.h header file, add the following member variables...
  - `float m_aspectRatio`
  - `float m_near, float m_far`
  - `glm::mat4 m_translationMatrix, m_perspectiveTransformation`

    - `glm::mat4 m_scaleMatrix, m_rotationMatrix`
    - `float m_thetaH, m_thetaW`
    - `glm::vec4 m_eye, m_up`
    - `glm::vec4 m_u, m_v, m_w`
- In the header file, also add definitions for the following helper methods...
    - `void CamtransCamera::updateProjectionMatrix()`
    - `void CamtransCamera::updatePerspectiveMatrix()`
    - `void CamtransCamera::updateScaleMatrix()`
    - `void CamtransCamera::updateViewMatrix()`
    - `void CamtransCamera::updateRotationMatrix()`
    - `void CamtransCamera::updateTranslationMatrix()`
- Define the above five methods in the bottom of `CamtransCamera.cpp`. You will fill these in later.
- Fill in the get methods in `CamtransCamera.cpp` to return the corresponding member variables.
    - `getScaleMatrix(), getPerspectiveMatrix()`
    - `getPosition()`
        - (think of this as the position of the camera, or in other words the ____)
    - `getLook(), getU(), getV(), getW()`
    - `getAspectRatio(), getHeightAngle()`
    - **You may need to add some of these yourself to the header and source file, especially to run the camtrans test suite. The ones provided are the ones used by the stencil code.**

Task 2:

- Fill in the empty methods you added to the CamtransCamera.cpp file for updating the **projection matrix**, **perspective matrix**, **view matrix**, **rotation matrix**, **translation matrix** and **scale matrix**.
    - These methods should update their respective matrices with whatever current information we have saved in our member variables. We will call them when we have made changes to the member variables pertaining to the matrix. This way we don't have to think too hard about what to change about the matrix itself; it will just refresh it entirely.
    - There are no **projection** and **view matrix** member variables in the stencil code, so in the `updateProjectionMatrix()` and `updatViewMatrix()` methods, you will need to call the update methods of the matrices that they are composed of.
    - For example, the projection matrix is composed of the scale and perspective transform, so this method should call `updateScaleMatrix()` and `updatePerspectiveMatrix()`.
    - Check out the algo to figure out what matrices should be updated when `updateViewMatrix()` is called.

Task 3:

- In `CamtransCamera::CamtransCamera` in the CamtransCamera.cpp file, orient the camera to be like the demo.
    - The **near clip plane** is 1 and the far clip plane is 30.
    - The **vertical view angle** is 60 degrees.
    - The screen **aspect ratio** is 1:1.
    - The **eye position** of the camera is at (2,2,2) in world coordinates.
    - The camera is **looking** at the origin, and its **up vector** is (0,1,0) in world coordinates.
- Fill in `CamtransCamera::setAspectRatio` to update the **aspect ratio** and call `updateProjectionMatrix()`.
- Now it's time to fill in `getProjectionMatrix()` and `getViewMatrix()`.
    - How would you calculate the projection and view matrices from our matrix member variables?
        - **Remember from the algo** that there were four matrices which formed the full camera matrix? That final camera matrix is just the product of the projection and view matrices. But which matrices belongs to which?
        - **Projection matrices** are those which modify the actual projection – matrices which are affected by changes in the view frustum (the six planes near, far, top, bottom, left, and right). The **view matrix** affects the positioning of the world within the camera's view.
- In `CamtransCamera::orientLook`, update the eye, look, and up variables with the new ones. Then recalculate **u**, **v**, and **w** based on those new values. Then you'll need to call `updateViewMatrix()` and `updateProjectionMatrix()`.
    - (Hint: `glm::normalize` might help here.)
- In `CamtransCamera::setHeightAngle`, update the **height angle** and **width angle**. Also be sure to call `updateProjectionMatrix()`. Calculation help here.
- In `CamtransCamera::translate`, you will need to move the **eye** and call `updateViewMatrix()`.
- In `CamtransCamera::rotateU`, `CamtransCamera::rotateV`, and `CamtransCamera::rotateW`, rotate around the respective vector by a degree value and then update the view matrix.
    - (Hint: Look at the algos section, and **remember to convert degrees to radians**.)
- In `CamtransCamera::setClip` set the **near** and **far planes,** and call `updateProjectionMatrix()`.

Your camera must support...

- maintaining matrices for the **projection transformation** and the **view transformation**.
- setting the camera's **absolute position** and orientation given an **eye position**, **look vector**, and **up vector**.
- setting the camera's **height angle** and **aspect ratio**.

- translating the camera in **world space**.
- rotating the camera about one of the axes in **its own virtual coordinate system**.
- setting the **near** and **far clipping planes**.
- Having the ability to, at any point, spit out the current **eye position**, **look vector**, **up vector**, **height angle** (in degrees), **aspect ratio**, or **world to film matrix**.

Testing

You can test your camera implementation by running your project and switching to the camtrans tab. Make sure the "Use orbit camera instead" box is not checked and you will be presented with your shapes scene using your Camtrans implementation. Compare the results of translations/rotations in your implementation with the results in the demo.

We have provided a **test suite** that you can use to test your CamtransCamera functionality. To use it,

- Make sure everything runs with the new changes that you want to test on your local machine.
- The test script on the department machines relies on an older Qt version, change the memcpy() call in line 93 in SupportCanvas2D.cpp from temp->sizeInBytes() to temp->byteCount().
- Push everything to your repo. Clone it into your folder on department machines through either FastX3 or ssh.
- Then run cs1230_camtrans_test from the directory containing your CS123.pro file.

End

Now you are ready to show your program to a TA to get checked off!
Be prepared to answer one or more of the following:

- Question 1: What is the difference between the projection matrix and the view matrix?
- Question 2: Question 3: Briefly explain what each matrix is responsible for doing and show a TA how each matrix is calculated in your code:
    - M1: Scaling the perspective
    - M2: Rotation
    - M3: Translation
    - M4: Original Point

Food for thought

Oh snap! Your camera can now display three-dimensional objects oriented in any way and viewed from any position! This lab will be used in future project assignments.