UNIVERSITÉ Concordia UNIVERSITY

COMP 371
Computer Graphics

Lab 03 – Basic shooter

Prepared by Nicolas Bergeron

UNIVERSITÉ Concordia UNIVERSITY

This Week

Tutorial:

Shoot projectiles

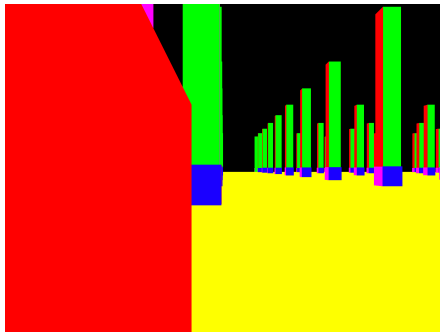
Models in view coordinates (always visible)

Implement simple cameras (1st and 3rd person)

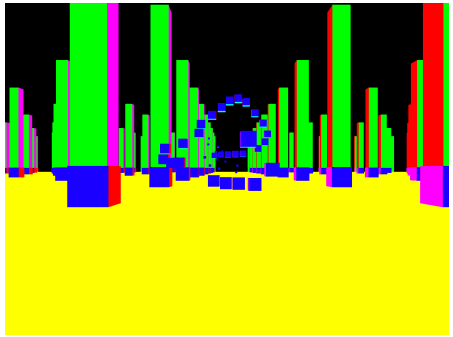
Exercises

Implement camera on rails

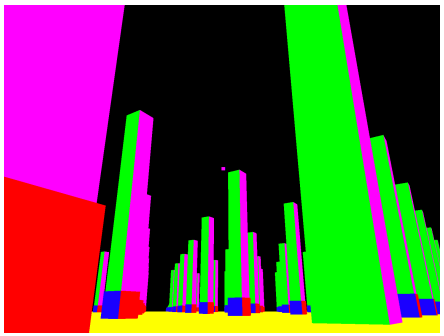
Expected results



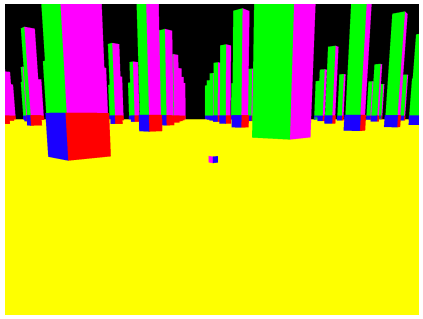
Depth Test



Projectiles



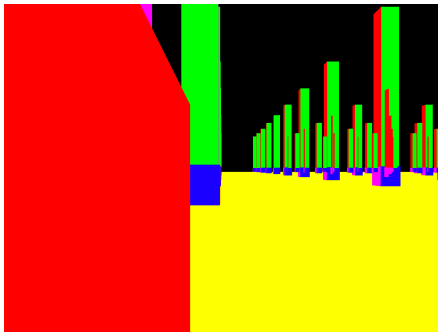
1st Person Camera



3rd Person Camera

Getting Started

- Download Lab01.zip from Moodle
- Download Lab03.zip and add lab03.cpp to the project (Visual Studio or Xcode)
- After compiling and running the application, you should see the image below.

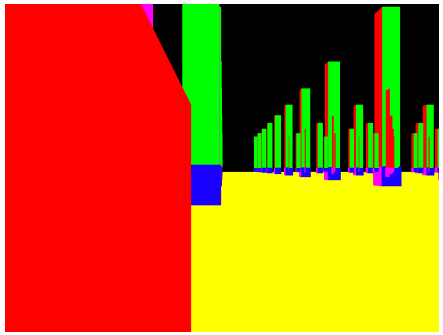


- Feel free to press A S D W to move the camera

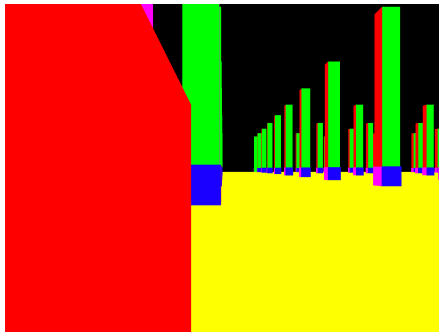
Outline for Lab03
Implement camera in OpenGL

- Shooting projectiles
- Drawing View Space (move with camera)
- Camera implementation
  - View vector represented in spherical coordinates
  - Spherical coordinates mapped to mouse inputs (Δx, Δy)
  - Keyboard inputs are moving camera position according to camera basis (A, S, D, W)
- 1st and 3rd person camera implementation
- Misc
  - Enable Depth Test

TUTORIAL
SHOOTING AND MOVING



Enabling Depth Test TODO1



Before

After

- Notice the starting program is not rendered properly
  - What draws on top depends on the drawing order of models
  - Far objects may obstruct near objects
  - This can be fixed by enabling the Depth Test
  - OpenGL implements the Z-buffer algorithm (covered class ~8)
- In OpenGL, we need to enable the depth test, and clear the depth buffer bit at the beginning of each frame.

```
glEnable(GL_DEPTH_TEST); // @TODO 1

// Add the GL_DEPTH_BUFFER_BIT to glClear - TODO 1
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Shooting Projectiles

- A class projectile is provided to update and draw projectiles over time
  - Constructor takes initial position and velocity of projectile as parameters and shaderProgram id
  - Update method recalculates position every frame
  - Draw method renders the projectile at correct position
- In your program, there is a list of projectiles
  - Spawn projectiles when pressing left mouse button
  - In the main loop, update and draw projectiles every frame

Code for shooting Projectiles
TODO 2

Add projectile to list on left mouse press

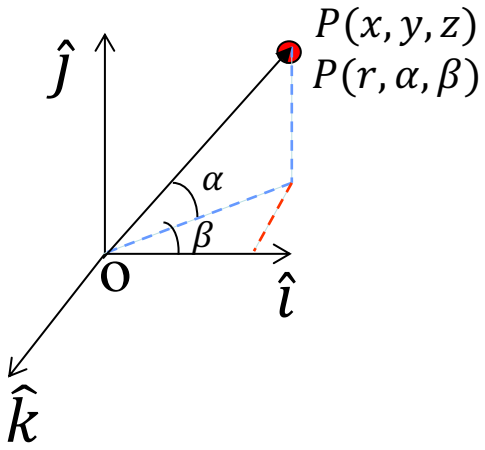
```
// @TODO2 - Shoot Projectiles
//
// shoot projectiles on mouse left click
// To detect onPress events, we need to check the last state and the current state to detect the state change
// Otherwise, you would shoot many projectiles on each mouse press
// ...

if (lastMouseLeftState == GLFW_RELEASE && glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS)
{
    const float projectileSpeed = 25.0f;
    projectileList.push_back(Projectile(cameraPosition, projectileSpeed * cameraLookAt, shaderProgram));
}
lastMouseLeftState = glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
```

Update and draw all projectiles

```
// @TODO 3 - Update and draw projectiles
// ...
for (list<Projectile>::iterator it = projectileList.begin(); it != projectileList.end(); ++it)
{
    it->Update(dt);
    it->Draw();
}
```

Spherical Coordinates

- Conversion between Cartesian and Spherical Coordinates



Parameters in Spherical Coordinates

α: Latitude / Pitch / Vertical Angle
β: Azimuth / Yaw / Horizontal Angle
r: Radius

Conversion Table

$$x = r \cos\alpha \cos\beta$$
$$y = r \sin\alpha$$
$$z = -r \cos\alpha \sin\beta$$

$$r = \sqrt{x^2 + y^2 + z^2}$$
$$\alpha = arc\sin\left(\frac{y}{r}\right)$$
$$\beta = arc\tan\left(\frac{-z}{x}\right)$$

Camera Implementation Using Spherical Coordinates

- Keep Track of Spherical Coordinates Angles (α, β)
- Convert Spherical Coordinates (r, α, β) to Cartesian Coordinates to determine the Camera $\overrightarrow{lookAt}$, then use cross products to set $\overrightarrow{Right}$ and $\overrightarrow{Up}$
- Key Press Changes Camera Position
  - [A] [D]: Move Camera along $\overrightarrow{Right}$
  - [W] [S]: Move Camera along $\overrightarrow{lookAt}$
- Mouse movement changes the Camera Direction
  - Δx : Affects angle β within $[-180°, 180°]$
  - Δy : Affects angle α within $[-85°, 85°]$

Camera Implementation Using Spherical Coordinates

- 1st Person Camera
  - The Camera Position is at the Center of the Sphere
- 3rd Person Camera
  - The Sphere is Centered on the Object of Interest (Projectile launcher)
  - The Camera Position is at the Surface of the Sphere looking at the Camera Center
- Additional Notes:
  - If the Camera is Controlling an Object on a Plane, the Camera Movement with [W] and [S] is along $\overrightarrow{lookAt}$ projected on that plane

1st and 3rd implementation (1/2)

- Disable mouse cursor after creating window (TODO 3)
  - `glfwSetInputMode(spWindow, GLFW_CURSOR, GLFW_CURSOR_DISABLED);`
- Calculate mouse motion and spherical coordinates to set lookAt vector
  - Just un-comment code for TODO 4
  - Read code line by line to understand it
  - Vertical angle is clamped in [-85, 85] to prevent it from being aligned with up vector, that would cause problems to calculate side vector

1st and 3rd implementation (2/2) TODO 5

- Move camera position (1st person) or sphere position (3rd person) along lookAt and side vector with A S D W

```
// @TODO 5 = use camera lookat and side vectors to update positions with ASDW
if (glfwGetKey(window, GLFW_KEY_W ) == GLFW_PRESS)
{
    cameraPosition += cameraLookAt * dt * currentCameraSpeed;
}

if (glfwGetKey(window, GLFW_KEY_S ) == GLFW_PRESS)
{
    cameraPosition -= cameraLookAt * dt * currentCameraSpeed;
}

if (glfwGetKey(window, GLFW_KEY_D ) == GLFW_PRESS)
{
    cameraPosition += cameraSideVector * dt * currentCameraSpeed;
}

if (glfwGetKey(window, GLFW_KEY_A ) == GLFW_PRESS)
{
    cameraPosition -= cameraSideVector * dt * currentCameraSpeed;
}
```

1st Person Implementation

- This should already be done. By setting the lookAt vector from mouse inputs, you can look around by moving the mouse.
- By moving the camera position with ASDW, you can navigate in the world.
- You can toggle between first person camera and third person camera by pressing keys 1 and 2 which sets the boolena flags `cameraFirstPerson`
- [Third person camera to be implemented next]

3rd Person Camera implementation TODO 6

```
// TODO 6
// Set the view matrix for first and third person cameras
// - In first person, camera lookat is set like below
// - In third person, camera position is on a sphere looking towards center

mat4 viewMatrix(1.0f);

if (cameraFirstPerson)
{
    viewMatrix = lookAt(cameraPosition, cameraPosition + cameraLookAt, cameraUp );
}
else
{
    // Position of the camera is on the sphere looking at the point of interest (cameraPosition)
    float radius = 5.0f;
    vec3 position = cameraPosition - vec3(radius * cosf(phi)*cosf(theta),
                                          radius * sinf(phi),
                                          -radius * cosf(phi)*sinf(theta));

    viewMatrix = lookAt(position, cameraPosition, cameraUp);
}

GLuint viewMatrixLocation = glGetUniformLocation(shaderProgram, "viewMatrix");
glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &viewMatrix[0][0]);
```

Drawing in View Space

- The world contains a spinning cube around the first person camera
  - Currently invisible due to backface culling
  - Disable backface culling if you are not convinced…
- We want this object to always draw in front of the camera (such as a weapon in a shooter game)
- We can render it in View Space
  - World matrix is identity
  - View matrix is relative to camera basis (such as a world matrix relative to world basis)

Spinning Cube in View space for 1st Person Camera (TODO 7)

```
// @TODO 7 - Draw in view space for first person camera
if (cameraFirstPerson)
{
    // Wolrd matrix is identity, but view transform like a world transform relative to camera basis
    // (1 unit in front of camera)
    //
    // This is similar to a weapon moving with camera in a shooter game
    mat4 spinningCubeWorldMatrix(1.0f);
    mat4 spinningCubeViewMatrix = translate(mat4(1.0f), vec3(0.0f, 0.0f, -1.0f)) *
                                  rotate(mat4(1.0f), radians(spinningCubeAngle), vec3(0.0f, 1.0f, 0.0f)) *
                                  scale(mat4(1.0f), vec3(0.01f, 0.01f, 0.01f));

    glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &spinningCubeWorldMatrix[0][0]);
    glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &spinningCubeViewMatrix[0][0]);
}
else
{
    // In third person view, let's draw the spinning cube in world space, like any other models
    mat4 spinningCubeWorldMatrix = translate(mat4(1.0f), cameraPosition) *
                                   rotate(mat4(1.0f), radians(spinningCubeAngle), vec3(0.0f, 1.0f, 0.0f)) *
                                   scale(mat4(1.0f), vec3(0.1f, 0.1f, 0.1f));

    glUniformMatrix4fv(worldMatrixLocation, 1, GL_FALSE, &spinningCubeWorldMatrix[0][0]);
}
glDrawArrays(GL_TRIANGLES, 0, 36);
```

EXERCISE

Assignment 1

- Start looking at assignment 1 if you haven't, try to understand the framework, look at shader files, look at scene files.
- Spheres and Box both inherit from model, which contains the data to set them in the world