COMP220: Graphics & Simulation

3: Vertices, Transforms & Projections

Worksheet Schedule

Worksheet	**Start**	**Formative deadline**
1: Framework	Week 2	Mon **15th Feb** 4pm (Week 4)
2: Basic scene	Week 4	Mon **1st Mar** 4pm (Week 6)
3: Plan/prototype	Week 6	Mon **15th Mar** 4pm (Week 8)
4: Final iteration	Week 8	Mon **12th Apr** 4pm (Week 10)

Learning outcomes

By the end of this week, you should be able to:

- **Recall** alternative ways to represent mesh vertices in memory.
- **Apply** basic transforms using the GLM library.
- **Explain** the constituents of the model-view-projection matrix and how it can be used to create a first-person camera controller.

Agenda

- Lecture (async):
  - **Compare** different ways to store vertex data in memory.
  - **Review** the transforms required to display 3D objects on a 2D screen.
- Workshop (sync):
  - **Adapt** our basic triangle implementation to draw meshes with multiple triangles efficiently.
  - **Experiment** with creating transforms using GLM and using them to move objects and the camera.

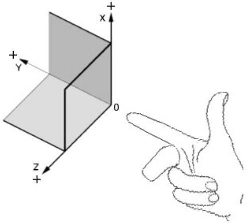
Schedule

16:00-16:10	Arrival, sign-in & overview
16:10-16:30	Demo & Exercise: Element Buffers
16:30-16:40	Introduction to GLM
16:40-17:10	Demo & Exercise: Transforms in GLM
17:10-17:30	Demo & Exercise: Model, View, Projection
17:30-18:00	Demo & Exercise: Camera Controller

Transformations in GLM

Right hand rule

OpenGL uses a **right-handed coordinate system**



- The *x*-**axis** points towards the **right-hand side** of the screen
- The *y*-**axis** points towards the **top** of the screen
- The *z*-**axis** points **out** of the screen

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**
- E.g. shifts them, rotates them, scales them, ...
- Many useful transformations can be **represented** by matrices
- Multiplying these matrices together **combines** the transformations
- Multiplying a vector by the matrix **applies** the transformation

GLM

- We will use the **GLM** library to do matrix calculations for us
- `http://glm.g-truc.net/`
- GLM aims to mirror GLSL data types (`vec4`, `mat4` etc) in C++
- Lets us perform calculations with vectors and matrices in C++
- GLM types can be passed into shaders as uniforms, e.g.

```
// transformLocation points to a uniform of type mat4
glm::mat4 transform = ...;
glUniformMatrix4fv(transformLocation, 1, GL_FALSE,
                   glm::value_ptr(transform));
```

Identity

The identity transformation does not change anything

```
// Default constructor for glm::mat4
// creates an identity matrix
glm::mat4 transform;
```

Translation

Translation shifts all points by the same vector offset

```
transform = glm::translate(transform,
                    glm::vec3(0.3f, 0.5f, 0.0f));
```

Scaling

Scaling moves all points closer or further from the origin by the same factor

```
transform = glm::scale(transform,
                glm::vec3(1.2f, 0.5f, 1.0f));
```

Rotation

- How do we represent a rotation in 3 dimensions?
- One way is by specifying the **axis** (as a vector) and the **angle** (in radians)
- Axis always runs through the origin

```
float angle = glm::pi<float>() * 0.5f;
glm::vec3 axis(0, 0, 1);
transform = glm::rotate(transform, angle, axis);
```

Combining transformations

```
transform = glm::translate(transform,
                    glm::vec3(0.5f, 0.5f, 0.0f));
transform = glm::rotate(transform, angle, axis);
```

- Transformations **do not commute** in general — changing the order will change the result
- The order they are applied is the **reverse** of what you might think — i.e. the above rotates **then** translates

Creating the model, view, projection matrix in GLM

The model matrix

Exactly what we've been doing so far today...

The view matrix

Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

```
glm::mat4 view = glm::lookAt(
  glm::vec3(2, 0, 2),    // eye
  glm::vec3(0, 0, 0),    // centre
  glm::vec3 up(0, 1, 0)  // up
);
```

- `eye` is the position of the camera
- `centre` is a point for the camera to look at
- `up` is which direction is "up" for the camera (usually the positive y-axis)

The projection matrix

```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

- **Field of view (FOV)**: how "wide" or "narrow" the view is
- **Aspect ratio**: should be `screenWidth / screenHeight`
- **Near and far clip planes**: fragments that fall outside this range of distances from the camera are not drawn

Also available: `glm::ortho` for orthographic projection

First person camera control

The plan

- Represent the player's **position** by a 3D vector
- Represent the player's **orientation** by Euler angles
- Mouse events change these angles
- View matrix is calculated using position and orientation
- To move forwards, use the Euler angles to find the "forward" vector, and offset the position by this vector

Keyboard and mouse in SDL

Use **relative mouse mode**:

```
SDL_SetRelativeMouseMode(SDL_TRUE);
```

- Hides the mouse pointer
- Prevents the mouse pointer from hitting the edge of the screen
- Gives us the distance the mouse has moved since last frame, rather than its current position

Use `SDL_GetKeyboardState` instead of handling individual keyboard events

- Allows us to check on every frame whether the key is held down
- Otherwise, the player will move jerkily according to the key repeat rate