MVD: Engine Programming

04 - Input and Debug Components

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Today, we're going to:

1) implement a free look camera, using a ControlSystem

2) Add some debug drawing to our scene, in order to help visualise stuff

Input

All 3D application manage their input via interaction with the underlying OS.

We are using a multiplatform API (GLFW) which abstracts all the platform-specific calls.

Documentation:

http://www.glfw.org/docs/latest/input_guide.html

Plugging it into the engine

We want to keep the engine as abstract as possible (within reason)

So in main.cpp we simply pass the important arguments to equivalent function in ‘our’ application (Game)

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    //quit
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, 1);
    GAME->key_callback(key, scancode, action, mods);
}

void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    GAME->mouse_button_callback(button, action, mods);
}
```

Updated Game.h

```
class Game
{
public:
    Game();
    void init();
    void update(float dt);

    //pass input straight to input system
    void updateMousePosition(int new_x, int new_y) {
        control_system_.updateMousePosition(new_x, new_y);
    }
    void key_callback(int key, int scancode, int action, int mods) {
        control_system_.key_mouse_callback(key, action, mods);
    }
    void mouse_button_callback(int button, int action, int mods) {
        control_system_.key_mouse_callback(button, action, mods);
    }

private:
    GraphicsSystem graphics_system_;
    ControlSystem control_system_;
    DebugSystem debug_system_;
};
```

Notes:

- we have created two new systems: DebugSystem and ControlSystem
- the input callbacks are passed straight through again to the ControlSystem.

Input: Mouse vs Keys/Buttons

Mouse:

- struct defined in ControlSystem.h, needs to store current position (*x* and *y*) but also **delta position** (difference since last frame) to track movement.

- declare instance of instance of struct in ControlSystem

- update struct member variables in callback function

Input: Mouse vs Keys/Buttons

Keys/buttons

- declare bool array for all keys buttons as member variable in ControlSystem

```
bool input[GLFW_KEY_LAST];
```

- (glfw3.h) contains #define macros for all keys/buttons
- set bool to true or false depend in key callback, depending on action:

```
void ControlSystem::key_mouse_callback(int key_button, int action, int mods) {

    if (action == GLFW_PRESS) input[key_button] = true;
    if (action == GLFW_RELEASE) input[key_button] = false;
}
```

Control Component?

Do we need a Control Component in our ECS?

We need to abstract common control mechanisms/camera-movement code, and keep it in the ControlSystem.

What are common control types?

enumerate ControlTypes

```
enum ControlType {
    ControlTypeFree,
    ControlTypeFPS,
    ControlTypeOrbit
};
```

ControlSystem has a member variable of type ControlType.

In ControlSystem::update(), we check to see which is the system's current ControlType, then call relevant function:

```
void ControlSystem::update(float dt) {
    if (control_type == ControlTypeFree) {
        updateFree(dt);
    }
}
```

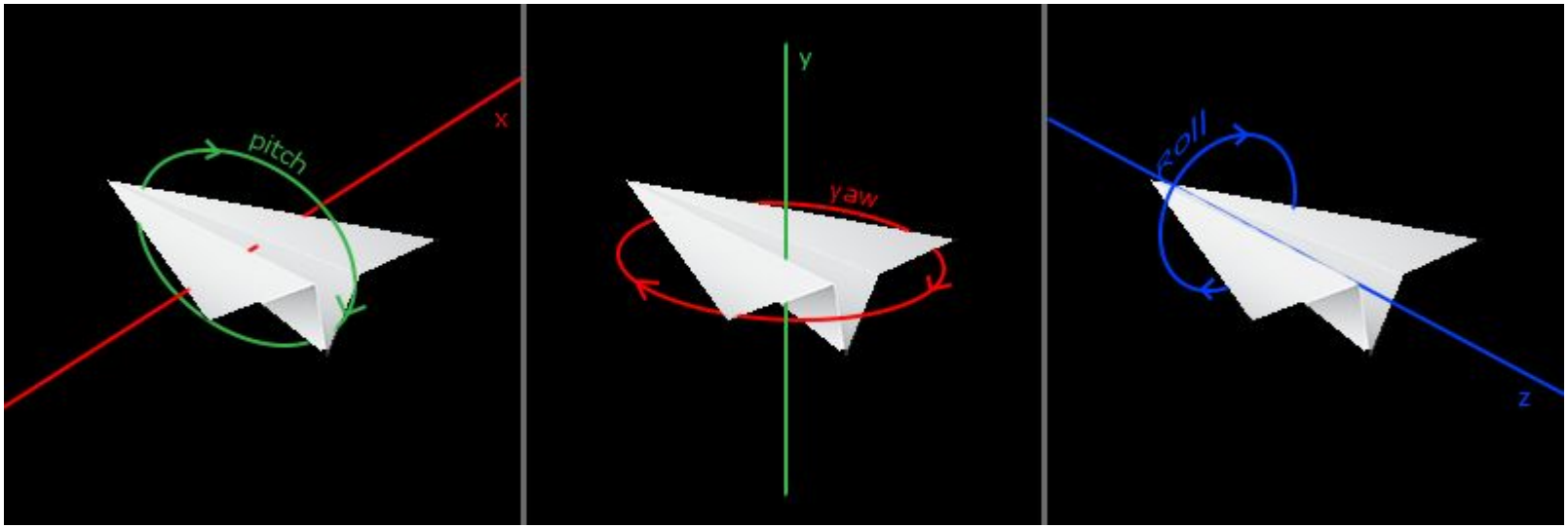
Free Camera movement

Move camera using WASD keys, along camera axes

- WS move along camera.forward
- AD move along camera.side
  - side = forward x (0,1,0) (x = cross product)

Free Camera look

We can rotate in three dimensions



But most free look cameras only permit pitch and yaw

Free Camera look

We need to rotate our **camera forward axis**, twice:

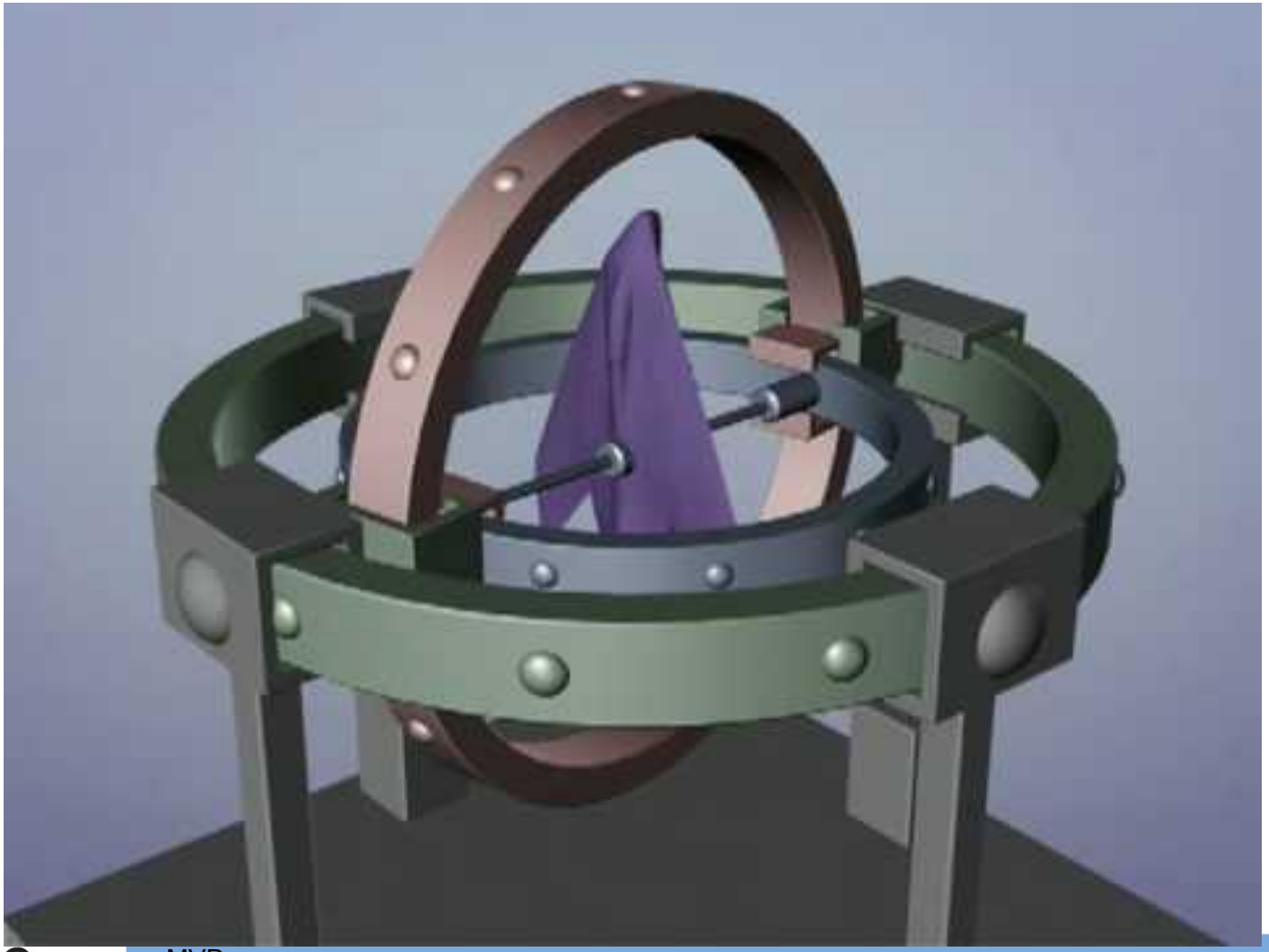
i) yaw (around y axis)

ii) pitch (around x axis)

1st rotation:

```
lm::mat4 R_yaw;
R_yaw.makeRotationMatrix(mouse_.delta_x * 0.005f, lm::vec3(0, 1, 0));
camera.forward = R_yaw * camera.forward;
```

Gimbal lock explained



Free camera look

2nd rotation:

To avoid Gimbal lock, we need to find the correct axis, which is the camera side vector (after first rotation)

```
lm::vec3 pitch_axis = camera.forward.normalize().cross(lm::vec3(0, 1, 0));
```

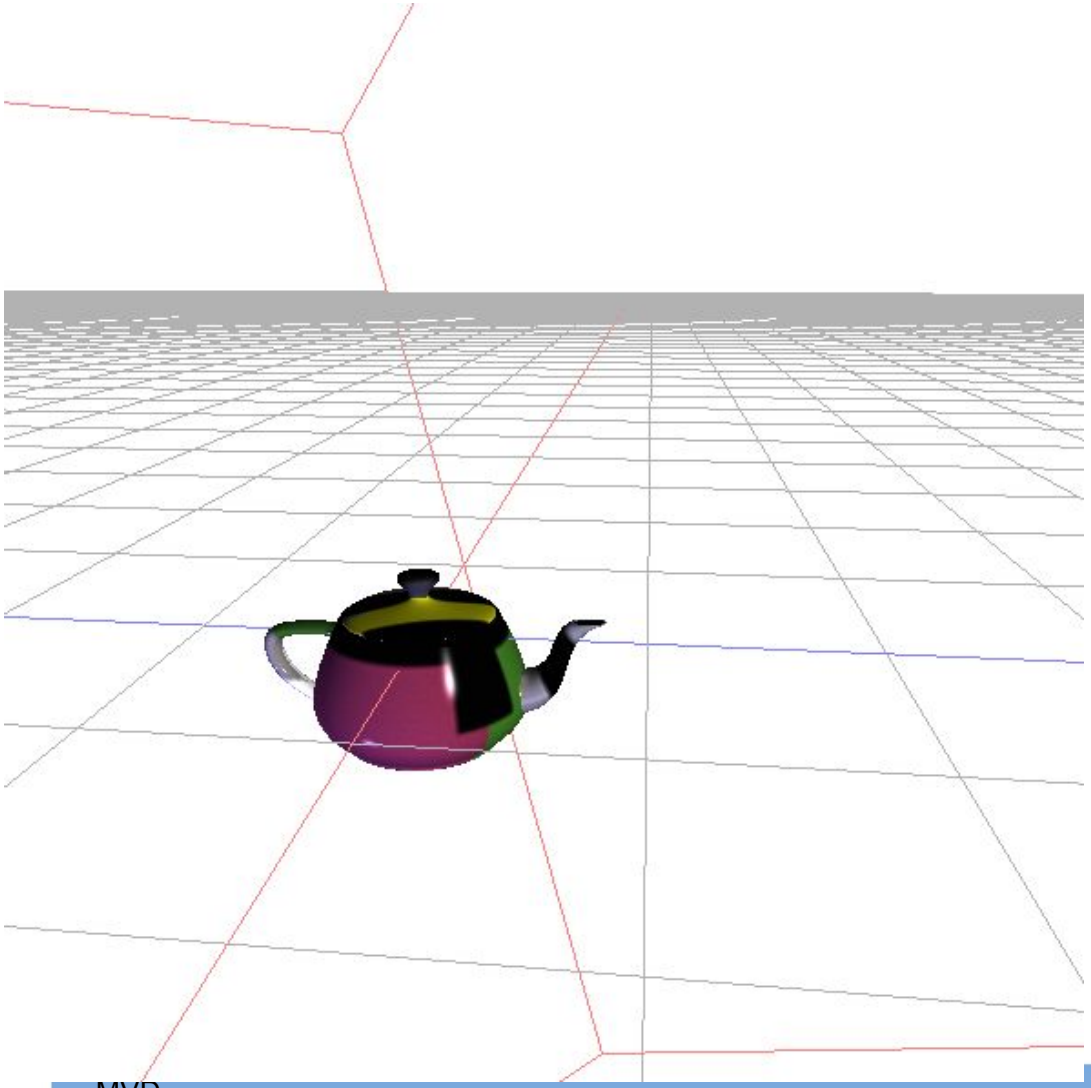
now rotate forward vector around this axis

Today task # 1

Implement an Free look camera:

- rotate camera with mouse, when left button down
- use WASD to move along camera orientation axes

Debug System



Debug System

The purpose of the debug system is to help the programmer

We draw simple lines and icons on the screen to show the position/dimensions of things that otherwise might be invisible.

Can you think of things that are useful to draw on screen?

Debug System

The DebugSystem should draw as fast as possible, but it will still use resources.

In a release build of the game you would probably remove the entire System using preprocesser directives.

For now we just activate/deactivate it will a simple boolean

Coloured line drawing

The **simplest** way of drawing coloured line is to create a colour buffer that matches the vertex buffer:

```
const GLfloat vertex_buffer_data[] = {
   10.0f, 14.0f, 5.0f,
   … };

const GLfloat colour_buffer_data[] = {
   1.0f, 0.0f, 0.0f,
   … };
```

Memory efficient line drawing

Each vertex contains a 4th component which is an index into an array of colours:

```
const GLfloat vertex_buffer_data[] = {
    10.0f, 14.0f, 5.0f, 1.0
//…
};
const GLfloat grid_colors[12] = {
    0.7f, 0.7f, 0.7f, //grey
    1.0f, 0.5f, 0.5f, //red
    0.5f, 1.0f, 0.5f, //green
    0.5f, 0.5f, 1.0f }; //blue
```

Line vertex shader

```
layout(location = 0) in vec4 a_vertex;
uniform mat4 u_mvp;
uniform vec3 u_color[4];
uniform int u_color_mod;

out lowp vec4 v_color;

void main() {;
    gl_Position = u_mvp * vec4(a_vertex.xyz, 1);

    v_color = vec4(u_color[ int(a_vertex.w) + u_color_mod ], 1.0f);
}
```

We will never change this shader, it is a fixed part of the engine, so we can define it as a const string in the .h

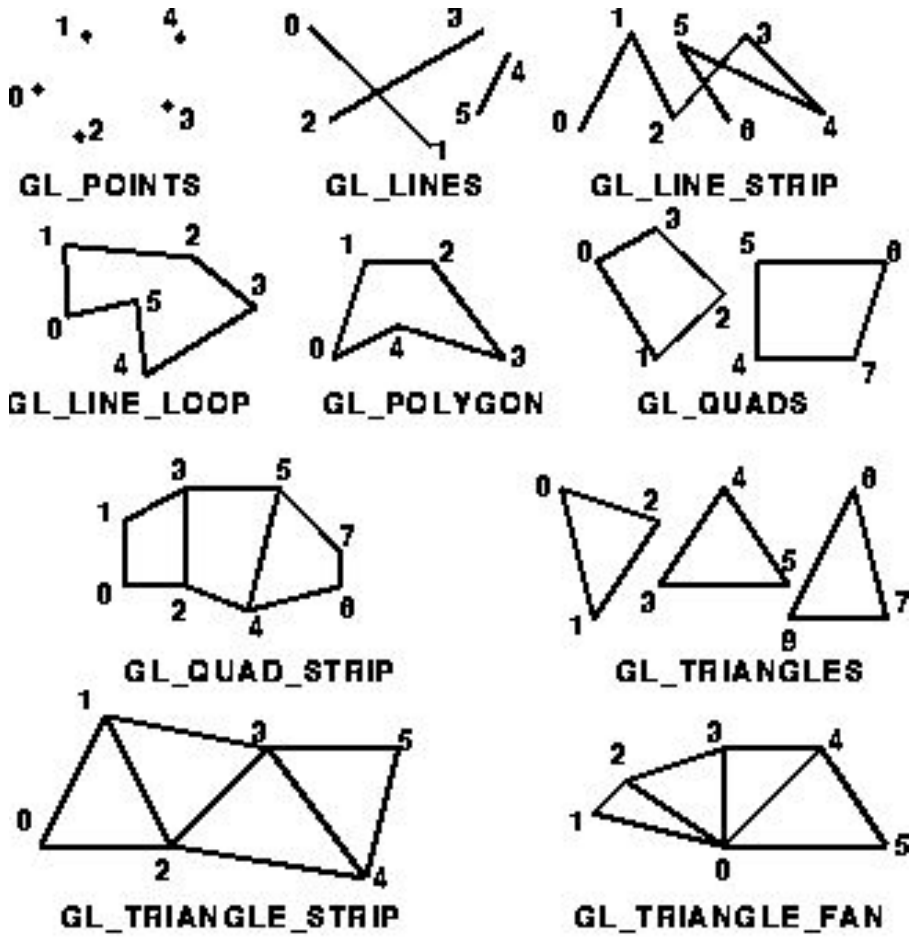
Creating a grid

```
for (num_divisions):
    add vertex zMin
    add vertex zMax

    add vertex xMin
    add vertex xMax

    if (num_divisions = num_divisions/2)
        change colour of line (4th component)
```

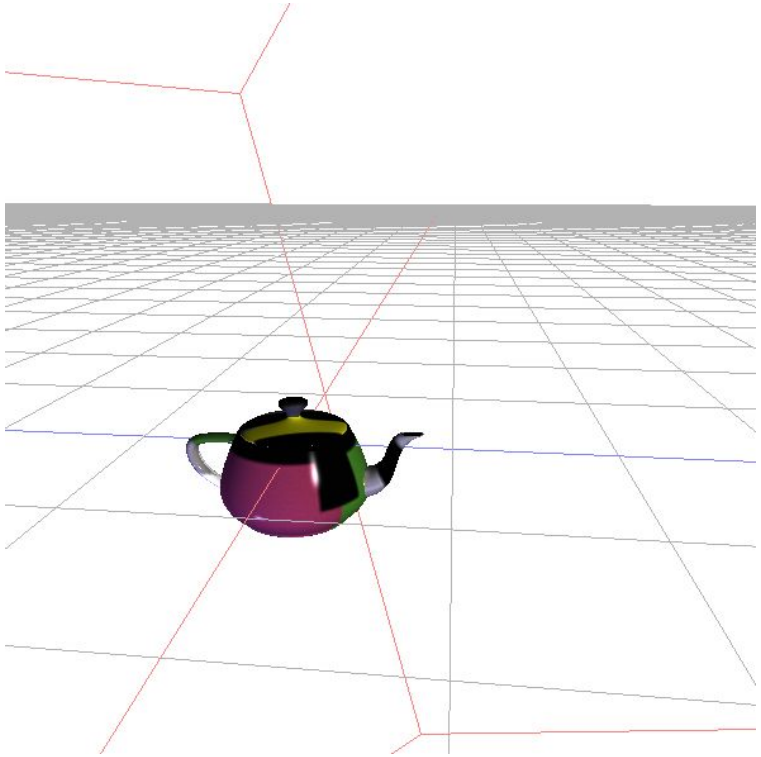
drawing lines



glDrawElements(GL_LINES, <#indices>, GL_UNSIGNED_INT, 0)

Today task 2

Draw grid_vao_ using grid_shader_ in DebugSystem



Drawing a frustum

Useful when using a debug camera to see view of main camera

Super useful when drawing a directional spotlight for shadowmapping

What are dimensions and coordinates of a frustum in OpenGL?

Transforming the frustum to clip space of current camera

The viewprojection matrix transforms a camera position/orientation/projection into NDC (clip space)

The **inverse of that matrix** bring the frustum back into world space

What are the dimensions of the frustum

So draw a frustum, we multiply the inverse vp matrix of the frustum we want to draw, with the vp matrix of our rendering camera:

```
lm::mat4 mvp = vp * cam_inv_vp;
```

Today task 3

There are two cameras defined in the scene

Debug System has a cube of lines, created in CreateCube_

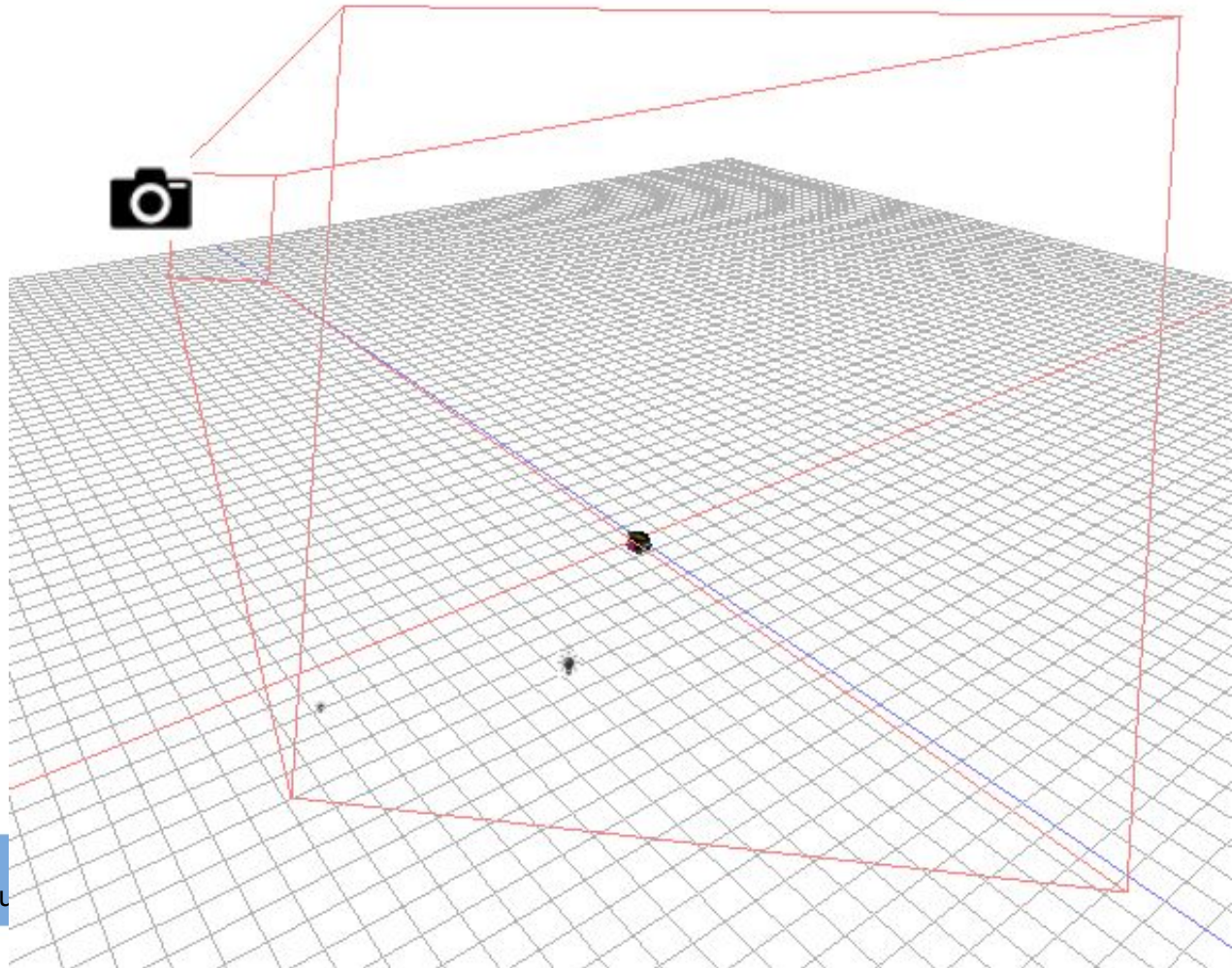
Use these lines to draw the frustum of each camera

where will the frustum of the ‘main’ camera be seen?

Adding icons

Icons are useful for invisible items in the seen

They are a simple textured quad, rendered to always face camera



Simple billboard

A billboard is a textured quad which always faces the camera.

'Real' billboards (using illumination) need to be properly aligned in world space, according to camera axes

'Simple' textured billboards are easier - remove rotation component from MVP matrix, so billboard always faces the camera

```
lm::mat4 bill_matrix;
for (int i = 12; i < 16; i++)
    bill_matrix.m[i] = mvp_matrix.m[i];
```

Today's task 4 :Drawing billboards

Load icon textures and shader in DebugSystem.init

in update()

For each camera component and each light component, draw icon!

Today

Create Free movement camera

Draw grid

Draw frusta

Draw icons