TRANSFORMATIONS & GAME OBJECTS

Transformations

- When rendering primitive shapes, they don't always need to be rendered at the centre of the game world; they can be moved around and transformed as well.
- For all transformations, you will need to use *GLM*'s built in matrix classes and functions
- Below is some sample code to create a translation of **4** in x, **0** in y and **-5** in z :

```
glm::mat4 translate;
translate = glm::translate(translate, glm::vec3(4, 0, -5));
```

Transformations

- All transformations need to be multiplied with the built-in *model* matrix for them to have any effects on the game objects
- This built-in matrix is always accessed via the *GameObject* class :

```
glm::mat4 translate;
translate = glm::translate(translate, glm::vec3(4, 0, -5));

//apply translation to model matrix
GameObject::MultiplyMatrix(translate);
GameObject::ApplyMatrix();
```

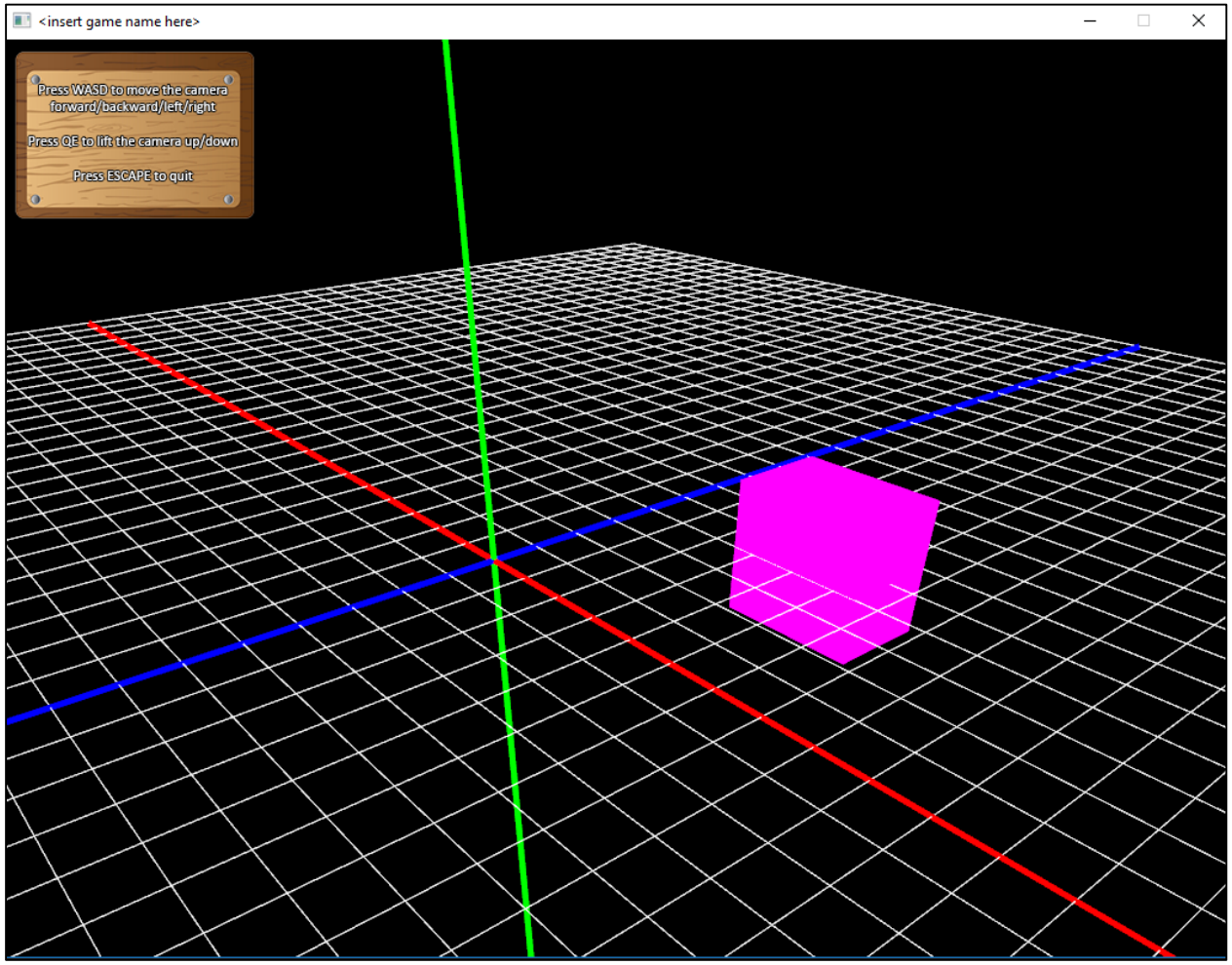
- ***Note : The Handmade engine supports 1 single model matrix***

Transformations

- Below is some sample code to create a **3x4x3** cube at world position **4** in x, **0** in y and **-5** in z :

```
//create translation transformation
glm::mat4 translate;
translate = glm::translate(translate, glm::vec3(4, 0, -5));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(translate);
GameObject::ApplyMatrix();

//render a purple cube at the set position
TheDebug::Instance()->DrawCube3D(3, 4, 2, Color::MAGENTA);
```

Transformations

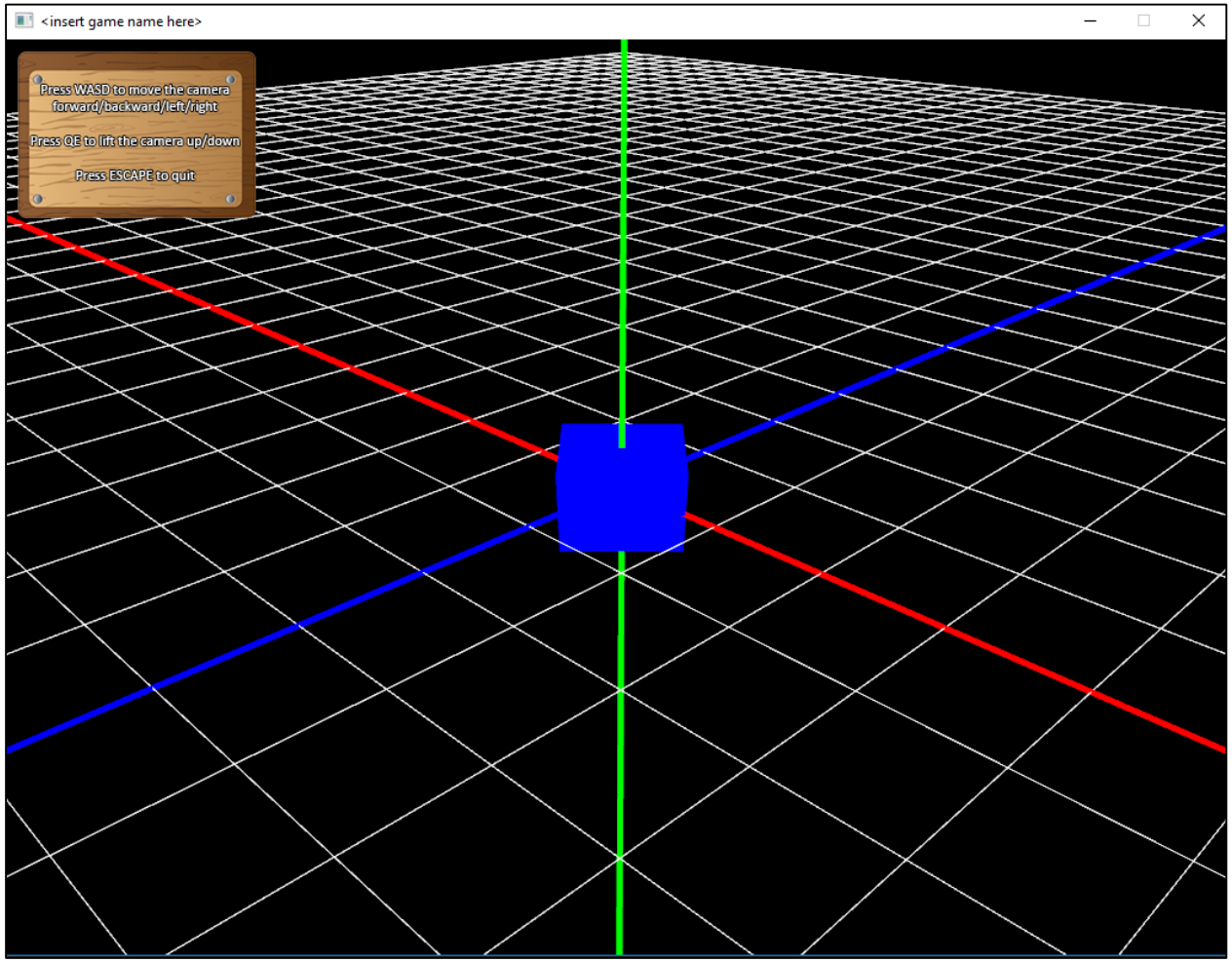


Transformations

- Below is some sample code to create a **1x1x1** cube with a **45** degree rotation around the y axis :

```
//create rotation transformation around Y axis
glm::mat4 rotate;
rotate = glm::rotate(rotate, glm::radians(45.0f), glm::vec3(0, 1, 0));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(rotate);
GameObject::ApplyMatrix();

//render a blue cube with the set rotation
TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::BLUE);
```

Transformations

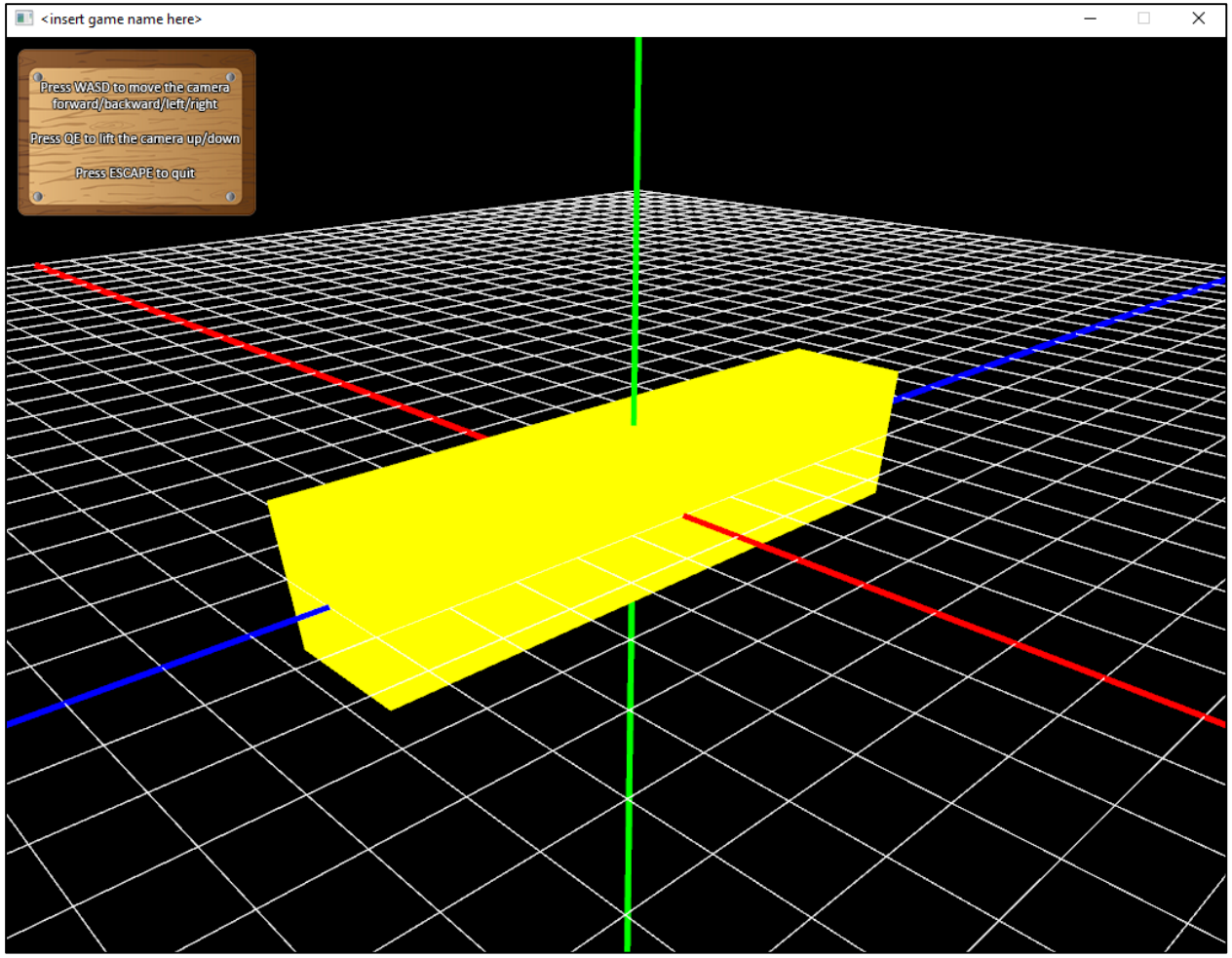


Transformations

- Below is some sample code to create a **1x1x1** cube with scaled value of **2** in x, **3** in y and **10** in z :

```
//create scaling transformation
glm::mat4 scale;
scale = glm::scale(scale, glm::vec3(2, 3, 10));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(scale);
GameObject::ApplyMatrix();

//render a scaled up yellow cube
TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::YELLOW);
```

Transformations



Transformations

- Objects can also be transformed by combining all transformations together, like so :

```
glm::mat4 translate;
glm::mat4 rotate;
glm::mat4 scale;
glm::mat4 totalTransform;

//move object into position
translate = glm::translate(translate, glm::vec3(4, 0, -5));

//rotate object around Y axis
rotate = glm::rotate(rotate, glm::radians(45.0f), glm::vec3(0, 1, 0));

//scale down object
scale = glm::scale(scale, glm::vec3(0.5, 0.5, 0.5));
```

- Now, we just need to accumulate all transformations together, like so :

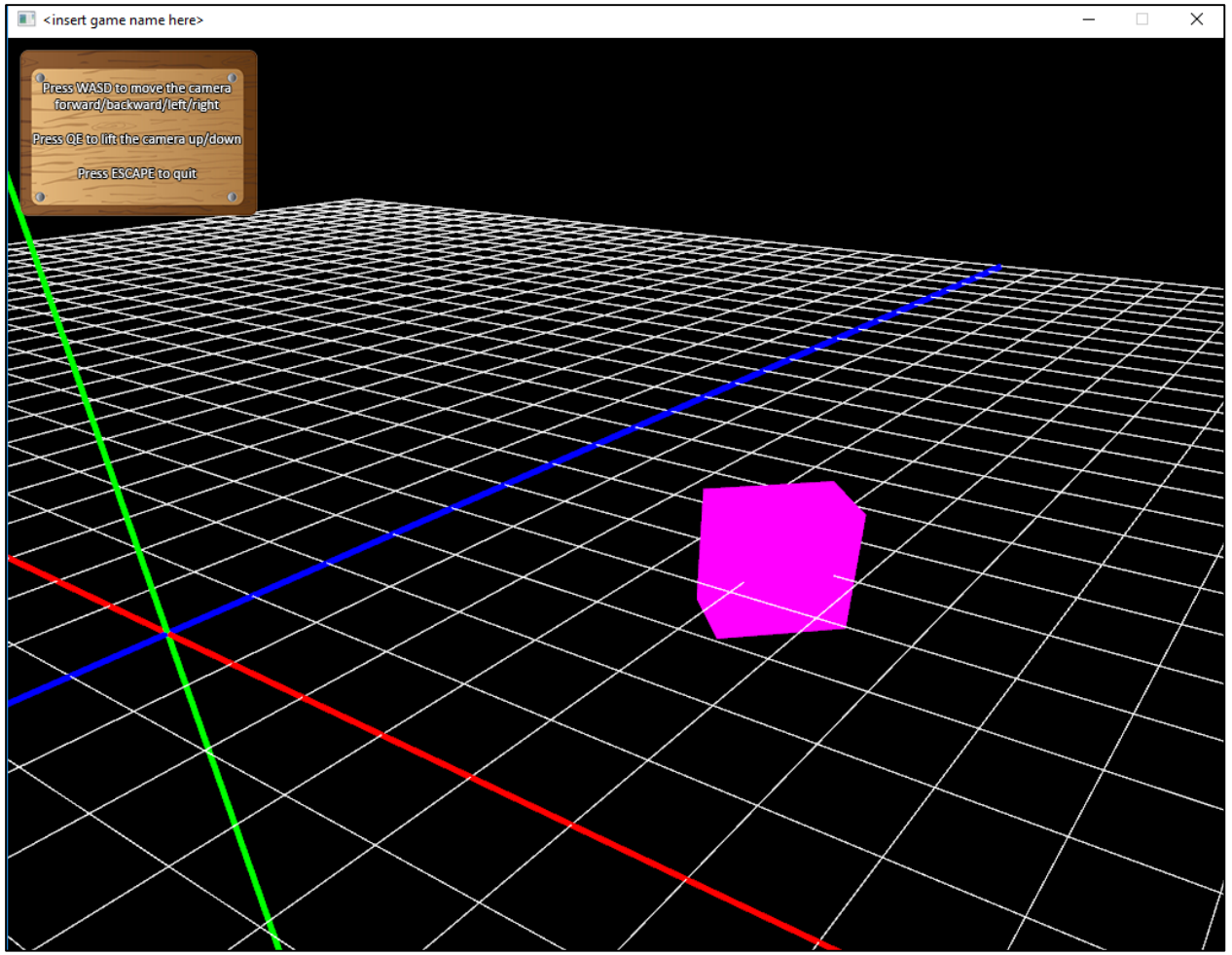
```
totalTransform = translate * rotate * scale;
```

Transformations

- One final thing left to do is apply these transformations to the *model* matrix and render the cube :

```
//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(totalTransform);
GameObject::ApplyMatrix();

//render a purple cube based on above transformation
TheDebug::Instance()->DrawCube3D(3, 4, 2, Color::MAGENTA);
```

Transformations



Transformations

- We can of course combine all of our transformations into **1** single matrix, like so :

```
glm::mat4 transform;

//move object into position
transform = glm::translate(transform, glm::vec3(4, 0, -5));

//rotate object around Y axis
transform = glm::rotate(transform, glm::radians(45.0f), glm::vec3(0, 1, 0));

//scale down object
transform = glm::scale(transform, glm::vec3(0.5, 0.5, 0.5));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(transform);
GameObject::ApplyMatrix();

//render a purple cube based on above transformation
TheDebug::Instance()->DrawCube3D(3, 4, 2, Color::MAGENTA);
```

Transformations

- You might want to apply different transformations to different objects.
- For this to work properly, we will need to reset the transformation matrix and the *model* matrix accordingly, like so :

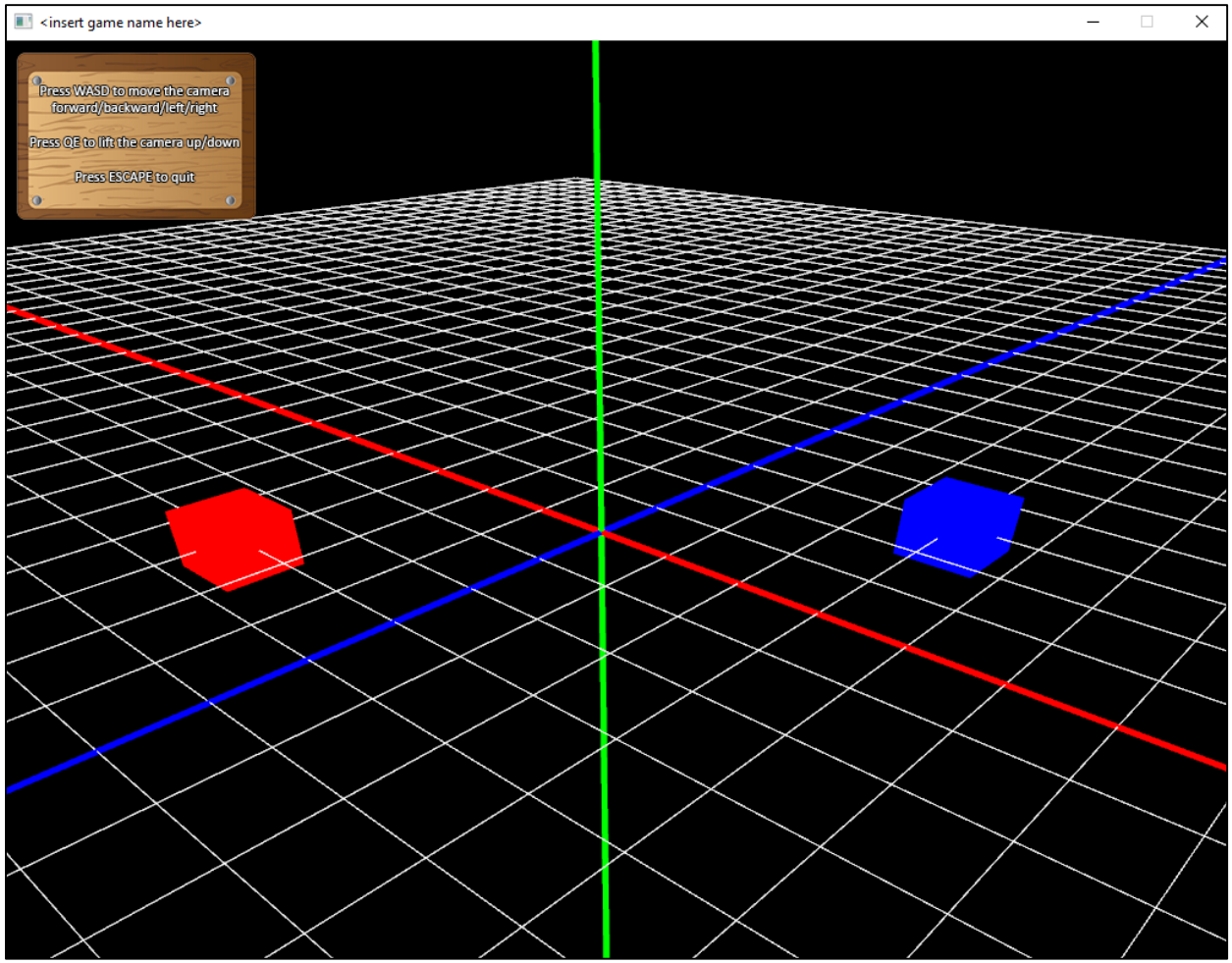
```
//reset transformation matrix
transform = glm::mat4(1.0f);

//reset model matrix
GameObject::SetIdentity();
```

- This needs to be done before every new transformation is made for every object in the scene
- Remember, transformations accumulate and the *GameObject* class only has **1** *model* matrix, so therefore it needs to be "flushed" before each different game object is rendered

Transformations

```
glm::mat4 transform;

//move object into position
transform = glm::translate(transform, glm::vec3(3, 0, -3));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(transform);
GameObject::ApplyMatrix();

//render a blue cube at set position
TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::BLUE);

//reset transformation matrix before applying a new transformation
transform = glm::mat4(1.0f);

//reset model matrix
GameObject::SetIdentity();

//move object into position
transform = glm::translate(transform, glm::vec3(-3, 0, 3));

//apply matrix transformation to model matrix
GameObject::MultiplyMatrix(transform);
GameObject::ApplyMatrix();

//render a red cube at set position
TheDebug::Instance()->DrawCube3D(1, 1, 1, Color::RED);
```

Transformations



Using Game Objects

- We would once again want all of our transformation and object code encapsulated in a class
- Again we will want to make use of the *GameObject* class to create individual game objects in the scene.
- Each object will have one or more matrices to represent the different transformations
- Every game object might also have a *position* variable (amongst others), so that we can always place our game object in the game world

Using Game Objects

- Each game object's *Update()* routine could contain the code to adjust our transformation matrix, like so :

```
void MyGameObject::Update()
{

    translate = glm::mat4(1.0f);
    translate = glm::translate(translate, glm::vec3(4, 0, -5));

}
```

Using Game Objects

- The *Draw()* function will contain the code to apply the matrix transformations to the *model* matrix :

```
bool MyGameObject::Draw()
{

    GameObject::MultiplyMatrix(translate);
    GameObject::ApplyMatrix();

    TheDebug::Instance()-> DrawCube3D(3, 4, 2, Color::MAGENTA);

    return true;

}
```

Adding Multiple Game Objects

- We can of course add more than one player object to the scene.
- Simply create a vector container of *Player* objects in the *MainState* header file :

```
std::vector<Player*> m_players;
```

Adding Multiple Game Objects

- Now, in the *MainState* source file, simply add as many players as you like.
- If you have added positional and color parameters in the *Player*'s constructor, you can then add them in different locations with a different color :

```
//create a few player objects
m_players.push_back(new Player(1.0, 0.0f, -2.0f, Color::RED));
m_players.push_back(new Player(-2.0, 0.0f, -4.0f, Color::BLUE));
m_players.push_back(new Player(3.0, 0.0f, -6.0f, Color::YELLOW));
m_players.push_back(new Player(-8.0, 0.0f, -8.0f, Color::GREEN));
```

Adding Multiple Game Objects

- We just need to adjust the *MainState's Update()* function so that it also loops through the vector of *Player* objects and updates them :

```
//loop through all player game objects in
//vector and update them only if they are active
for (auto it = m_players.begin(); it != m_players.end(); it++)
{
    if ((*it)->IsActive())
    {
        (*it)->Update();
    }
}
```

Adding Multiple Game Objects

- The same applies to the *MainState's Draw()* function - it needs to loop through the vector of *Player* objects and render them :

```
//loop through all player game objects in vector and
//display them only if they are active and visible
for (auto it = m_players.begin(); it != m_players.end(); it++)
{
    if ((*it)->IsActive() && (*it)->IsVisible())
    {
        (*it)->Draw();
    }
}
```

Adding Multiple Game Objects

- One final thing left, and that is to set the *model* matrix to the identity before each transformation is applied from within each game object :

```
bool Player::Draw()
{

    GameObject::SetIdentity();

    GameObject::MultiplyMatrix(m_translation);
    GameObject::ApplyMatrix();

    TheDebug::Instance()->DrawCube3D(m_size.x, m_size.y, m_size.z, m_color);

    return true;

}
```

Adding Multiple Game Objects