Game Engine Development II

Week3

Hooman Salamat

Scene Rendering

Objectives

- Examine entity systems in concept and practice
- Explore the viewable area of our world and scrolling
- Implement tree-based scene graphs, rendering and updating of many entities
- Implement the composition of all elements to shape the world

The `Entity` Class

- An entity represents some game element in the world
  - Other planes (friendly and enemy)
  - Projectiles (bullets and missiles)
  - Pickups
- Basically what the player can interact with
- Going to have a `velocity` attribute
- Let's see what the definition looks like...

The Entity Class

```
class Entity
{
public:
   void setVelocity(sf::Vector2f velocity);
   void setVelocity(float vx, float vy);
   sf::Vector2f getVelocity() const;
private:
   sf::Vector2f mVelocity;
};
```

- Vector2f has a default constructor that sets x and y to zero

The `Entity` Class

```
void Entity::setVelocity(sf::Vector2f velocity)
{
   mVelocity = velocity;
}

void Entity::setVelocity(float vx, float vy)
{
   mVelocity.x = vx;
   mVelocity.y = vy;
}

sf::Vector2f Entity::getVelocity() const
{
   return mVelocity;
}
```

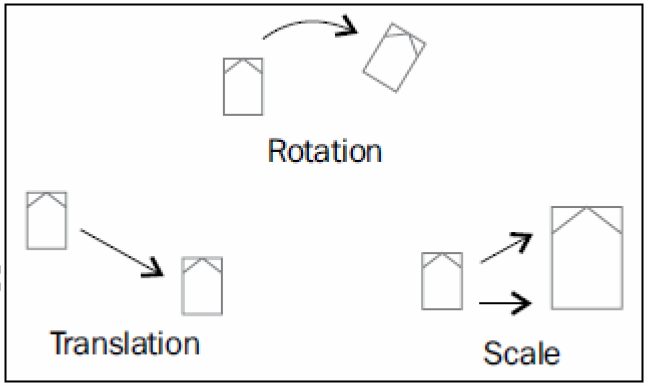
Aircraft Class

```
class Aircraft : public Entity
{
public:
    enum Type
    {
      Eagle,
      Raptor,
    };
public:
    explicit Aircraft(Type type);
private:
    Type mType;
};
```

Transforms

- A geometrical transform specifies the way an object is represented on screen
  - Translation -> position
  - Rotation -> orientation
  - Scale -> size

- SFML provides these in a class c sf::Transformable



sf::Transformable

- Accessors (getters/setters):
  - setPosition (), move(), rotate(), getScale()
  - setOrigin(), getOrigin()
- High-level classes such as `Sprite`, `Text` and `Shape` are derived from `Transformable` and `Drawable`

Sf::Drawable

- sf::Drawable is a stateless interface and provides only a pure virtual function with the following signature:

virtual void Drawable::draw(sf::RenderTarget& target,sf::RenderStates states) const = 0

- The first parameter specifies, where the drawable object is drawn to. Mostly, this will be a sf::RenderWindow. The second parameter contains additional information for the rendering process, such as blend mode (how pixel of the object are blened, transform (how the object is positioned/rotated/scaled), the used texture (what image is mapped to the object), or shader (what custom effectis applied to the object).
- SFML's high-level classes Sprite, Text, and Shape are all derived fromTransformable and Drawable.

Scene Graphs

- A scene graph is developed to transform the hierarchies
  - Consists of multiple nodes called scene nodes
  - Each node can store an object that is drawn
  - Represented by class called `SceneNode`
  - To store the children, we use vector container `std::vector<SceneNode>`

SceneNode Class

```
class SceneNode
{
public:
  typedef std::unique_ptr<SceneNode> Ptr;
public:
  SceneNode();
private:
  std::vector<Ptr> mChildren;
  SceneNode* mParent;
};
```

SceneNode Class

- We provide an interface to insert and remove child nodes:

```
void attachChild(Ptr child);
Ptr detachChild(const SceneNode& node);
```

SceneNode Class

```
void SceneNode::attachChild(Ptr child)
{
     child->mParent = this;
     mChildren.push_back(std::move(child));
}

SceneNode::Ptr SceneNode::detachChild(const SceneNode& node)
{
   auto found = std::find_if(mChildren.begin(), mChildren.end(),
   [&] (Ptr& p) -> bool { return p.get() == &node; });

   assert(found != mChildren.end());
   Ptr result = std::move(*found);
   result->mParent = nullptr;
   mChildren.erase(found);
   return result;
}
```

SceneNode Class Updated

```
class SceneNode : public sf::Transformable, public sf::Drawable,
                    private sf::NonCopyable
{
public:
   typedef std::unique_ptr<SceneNode> Ptr;
public:
   SceneNode();
   void attachChild(Ptr child);
   Ptr detachChild(const SceneNode& node);
private:
   virtual void draw(sf::RenderTarget& target,
                     sf::RenderStates states) const;
   virtual void drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const;
private:
   std::vector<Ptr> mChildren;
   SceneNode* mParent;
};
```

SceneNode Class Updated

- The class can then be used thus:

```
sf::RenderWindow window(...);
SceneNode::Ptr node(...);
window.draw(*node); // note: no node->draw(window) here!
```

Aircraft Revisited

```
class Aircraft : public Entity // inherits indirectly SceneNode
{
public:
   explicit Aircraft(Type type);
   virtual void drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const;
private:
   Type mType;
   sf::Sprite mSprite;
};

void Aircraft::drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const
{
   target.draw(mSprite, states);
}
```

Resetting the Origin

- By default, the origin of sprites is in their upper-left corner
    - For alignment or rotation, it might be better to work with their center, and we can set it thus:

```
sf::FloatRect bounds = mSprite.getLocalBounds();
mSprite.setOrigin(bounds.width / 2.f, bounds.height / 2.f);
```

Scene Layers

- Different nodes must be rendered in a certain order
  - Can't have ground above sky, for example
  - Common sense
  - UI as top layer

```
enum Layer
{
   Background,
   Air,
   LayerCount
};
```

Updating the Scene

- During an update, entities move and interact, collisions are checked and projectiles are launched
- We can add the following to `SceneNode`

```
public:
  void update(sf::Time dt);
private:
  virtual void updateCurrent(sf::Time dt);
  void updateChildren(sf::Time dt);
```

Updating the Scene

```
void SceneNode::update(sf::Time dt)
{
   updateCurrent(dt);
   updateChildren(dt);
}

void SceneNode::updateCurrent(sf::Time)
{
}

void SceneNode::updateChildren(sf::Time dt)
{
   FOREACH(Ptr& child, mChildren)
        child->update(dt);
}
```

Updating the Scene

- We also have to make the following additions to the `Entity` class:

```
private:
  virtual void updateCurrent(sf::Time dt);

...

void Entity::updateCurrent(sf::Time dt)
{
  move(mVelocity * dt.asSeconds());
}
```

Updating the Scene

- We also have to make the following additions to the `Entity` class:

```
private:
  virtual void updateCurrent(sf::Time dt);

...

void Entity::updateCurrent(sf::Time dt)
{
  move(mVelocity * dt.asSeconds());
}
```

Absolute Transforms

- In order to find out if two objects collide, we have to look at their world transform, not local or relative transforms
- We perform the following absolute transform:

```
sf::Transform SceneNode::getWorldTransform() const
{
   sf::Transform transform = sf::Transform::Identity;
   for (const SceneNode* node = this; node != nullptr;
        node = node->mParent)
        transform = node->getTransform() * transform;
   return transform;
}

sf::Vector2f SceneNode::getWorldPosition() const
{
   return getWorldTransform() * sf::Vector2f();
}
```

The View

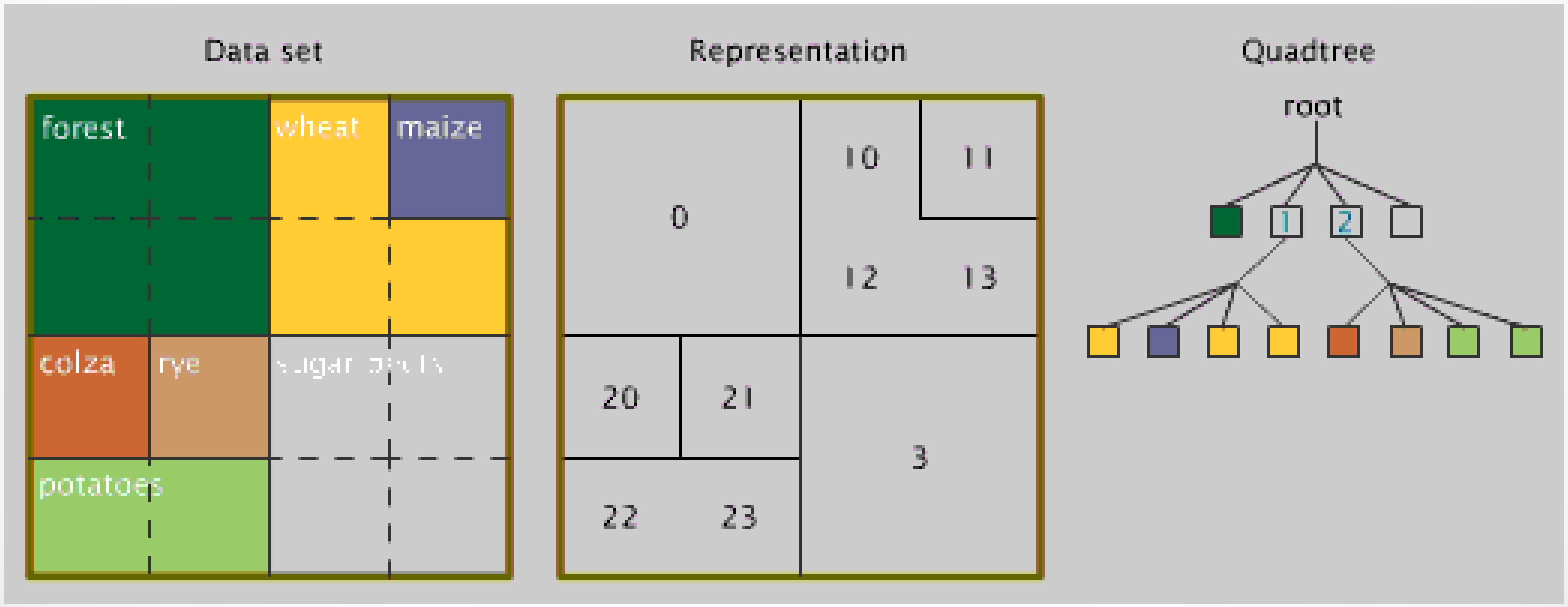
- In our case, a view is a rectangle that represents the subset of our world that we want to render at a particular time
- We are provided a class called `sf::View`
- With the view, we can scroll, zoom and rotate with ease
- Since our game's action occurs in a vertical corridor, we scroll the view at a constant speed towards the negative y
  - mView(0.f, -40 * dt.asSecond());
- sf::View::zoom(float factor) function to easily approach or move away from the center of the view
  - mView.zoom(0.2);
- sf::View::rotate(float degree) to add a rotation angle to the current one
- sf::View::setRotation(float degrees) to set the rotation of the view to an absolute value
  - mView.rotate(45);

View Optimization

- A Draw call is an expensive operation. We use culling to check if objects are within the viewing rectangle and then draw them.
- Game developers implement spatial subdivision: Dividing scene in multiple cells, which group all objects that reside within that given cell
- Cull a group of objects that are not in the view
- Quad Tree and Circle tree are two famous ways to subdivide space.

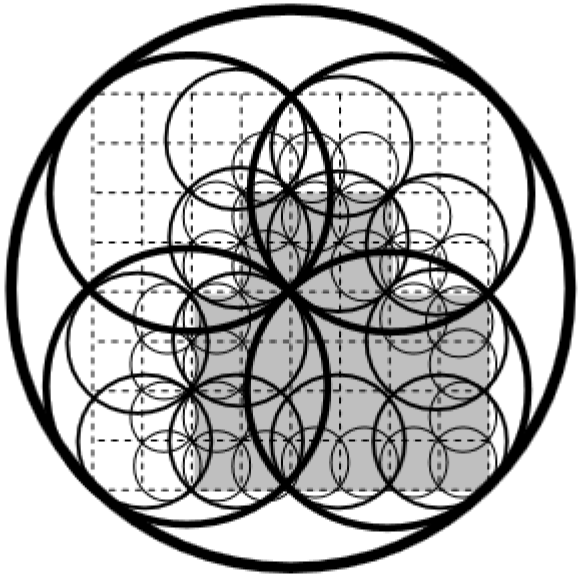
Quad Tree

- A quad tree is a hierarchical tree of cells. Only leaf nodes can contain objects and subdivide when a predetermined number of objects are present.



Circle Tree

- Similar to the quad tree, but instead each cell is a circle.
- Allows a different distribution of the objects



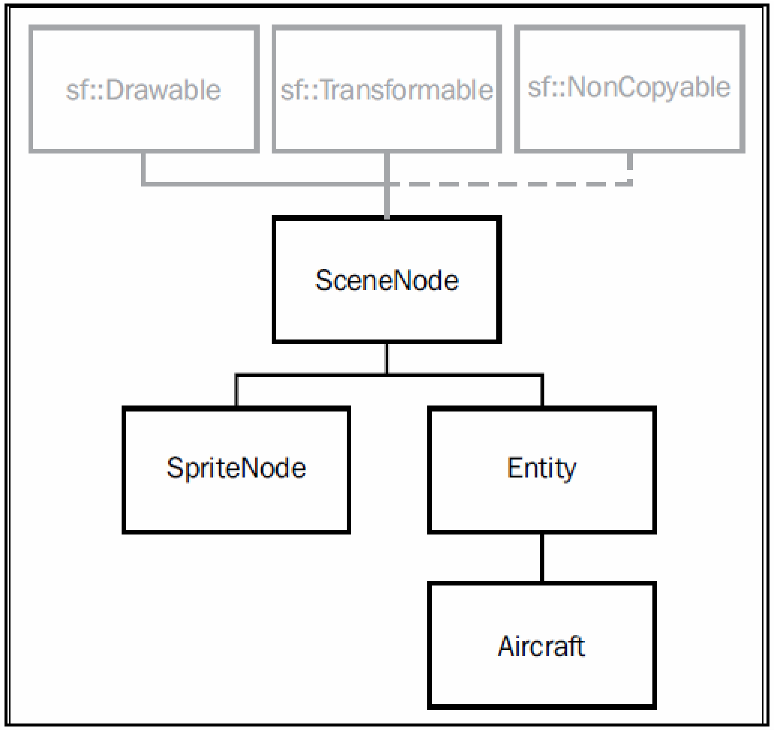
The `SpriteNode` Class

- The `SpriteNode` class represents a background sprite

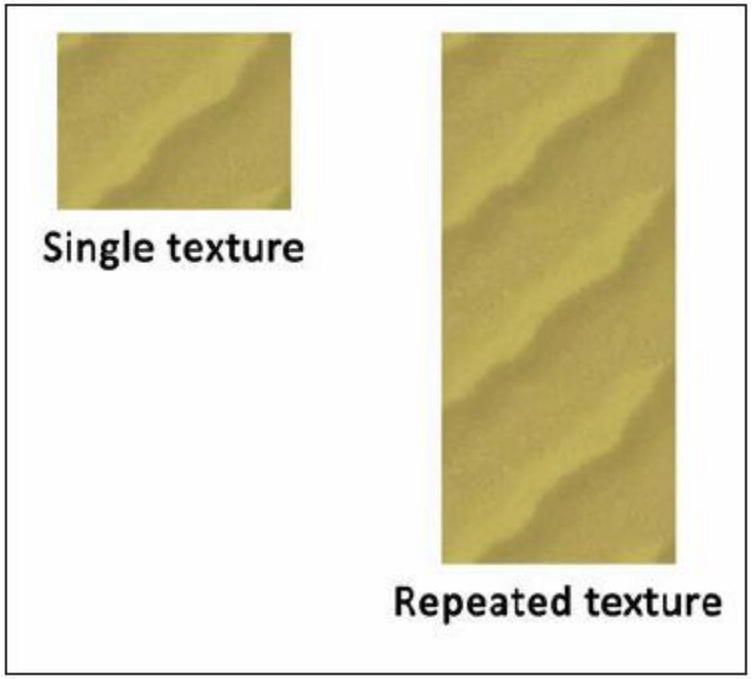
```
class SpriteNode : public SceneNode
{
public:
   explicit SpriteNode(const sf::Texture& texture);
   SpriteNode(const sf::Texture& texture, const sf::IntRect& rect);
private:
   virtual void drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const;
private:
   sf::Sprite mSprite;
};
```

The `SpriteNode` Class

- In the diagram below, the grey classes are part of SFML and the black ones are ours



Texture Repeating



- Every `sf::Texture` comes along with the option to enable repeating along both axis with the `sf::Texture::setRepeated(bool)` function

Composing the World

- The `World` class must contain all rendering data:
  - A reference to the render window
  - The world's current view
  - A texture holder with all the textures needed inside the world
  - The scene graph
  - Some pointers to access the scene graph's layer nodes
  - The bounding rectangle of the world, storing its dimensions
  - The position where the player's plane appears in the beginning
  - The speed with which the world is scrolled
  - A pointer to the player's aircraft

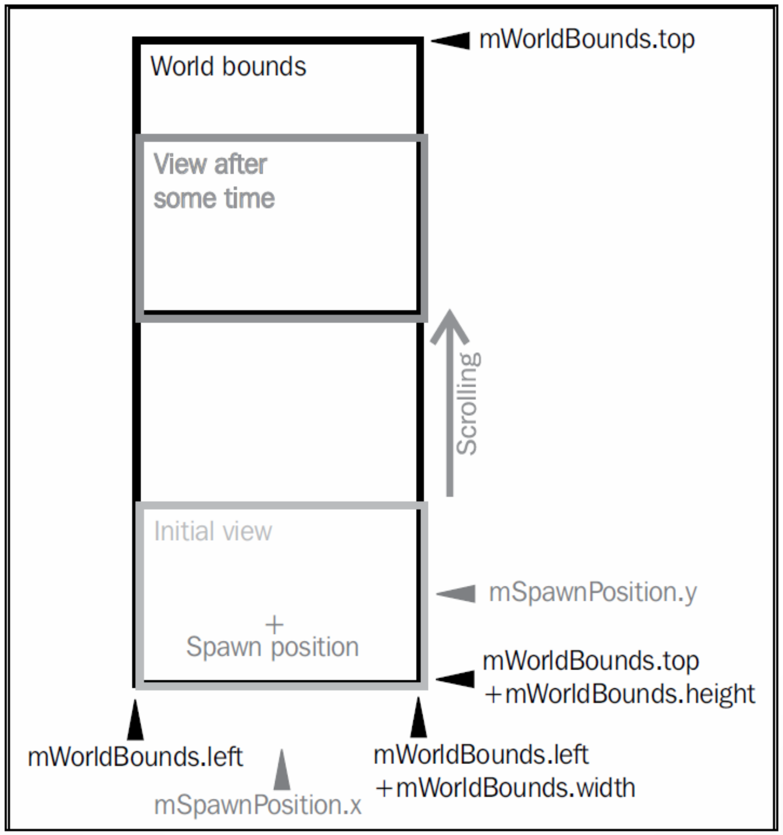
The `World` Class

```
class World : private
    sf::NonCopyable
{
public:
    explicit
    World(sf::RenderWindow&
    window);
    void update(sf::Time dt);
    void draw();
private:
    void loadTextures();
    void buildScene();
    private:
    enum Layer
    {
        Background,
        Air,
    LayerCount
    };
```

```
private:
    sf::RenderWindow& mWindow;
    sf::View mWorldView;
    TextureHolder mTextures;
    SceneNode mSceneGraph;
    std::array<SceneNode*, LayerCount>
    mSceneLayers;
    sf::FloatRect mWorldBounds;
    sf::Vector2f mSpawnPosition;
    float mScrollSpeed;
    Aircraft* mPlayerAircraft;
};
```

Composing the World

- The following diagram represents the world dimensions:



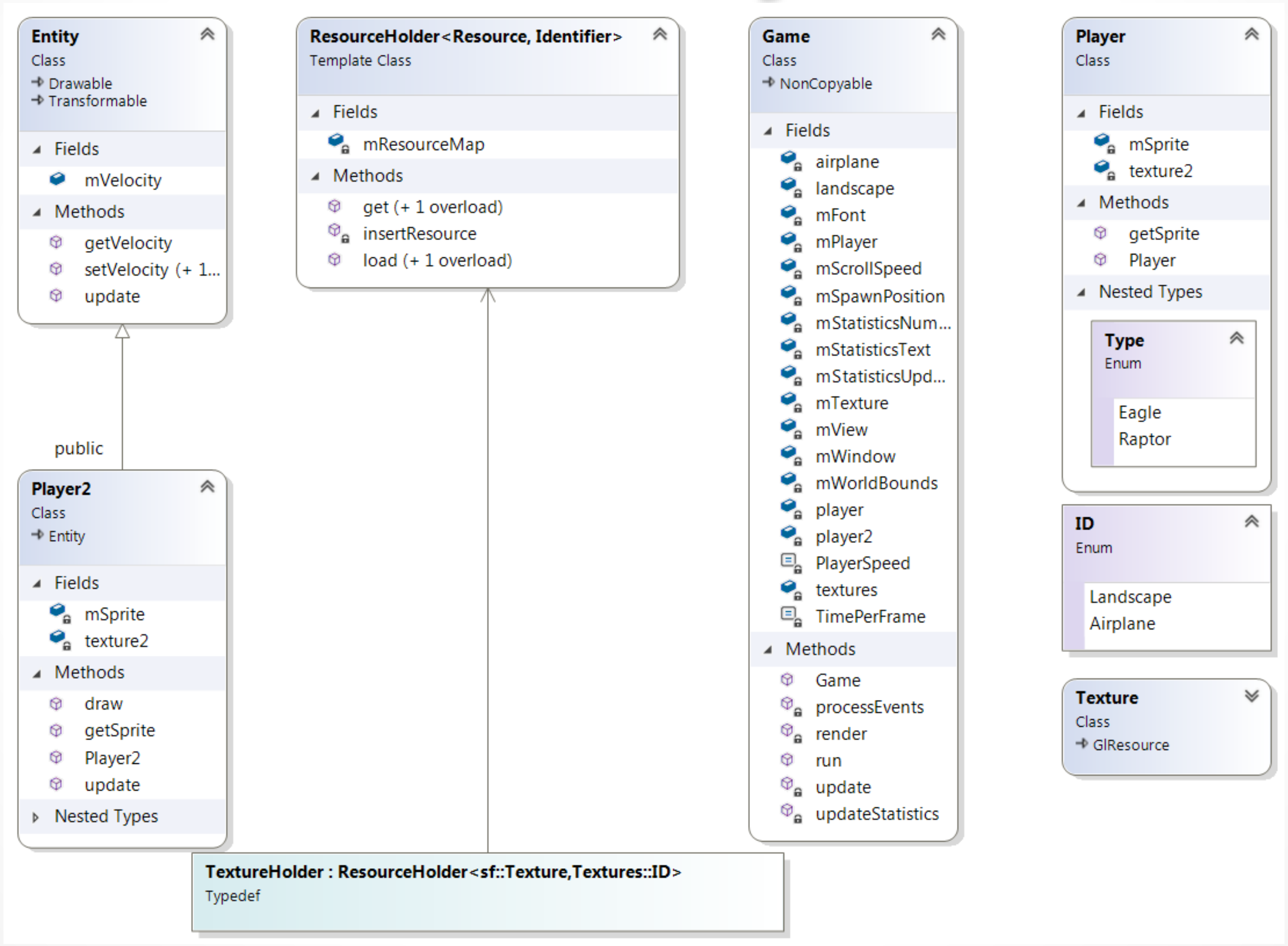
Loading the Textures

```
void World::loadTextures()
{
   mTextures.load(Textures::Eagle, "Media/Textures/Eagle.png");
   mTextures.load(Textures::Raptor, "Media/Textures/Raptor.png");
   mTextures.load(Textures::Desert, "Media/Textures/Desert.png");
}
```

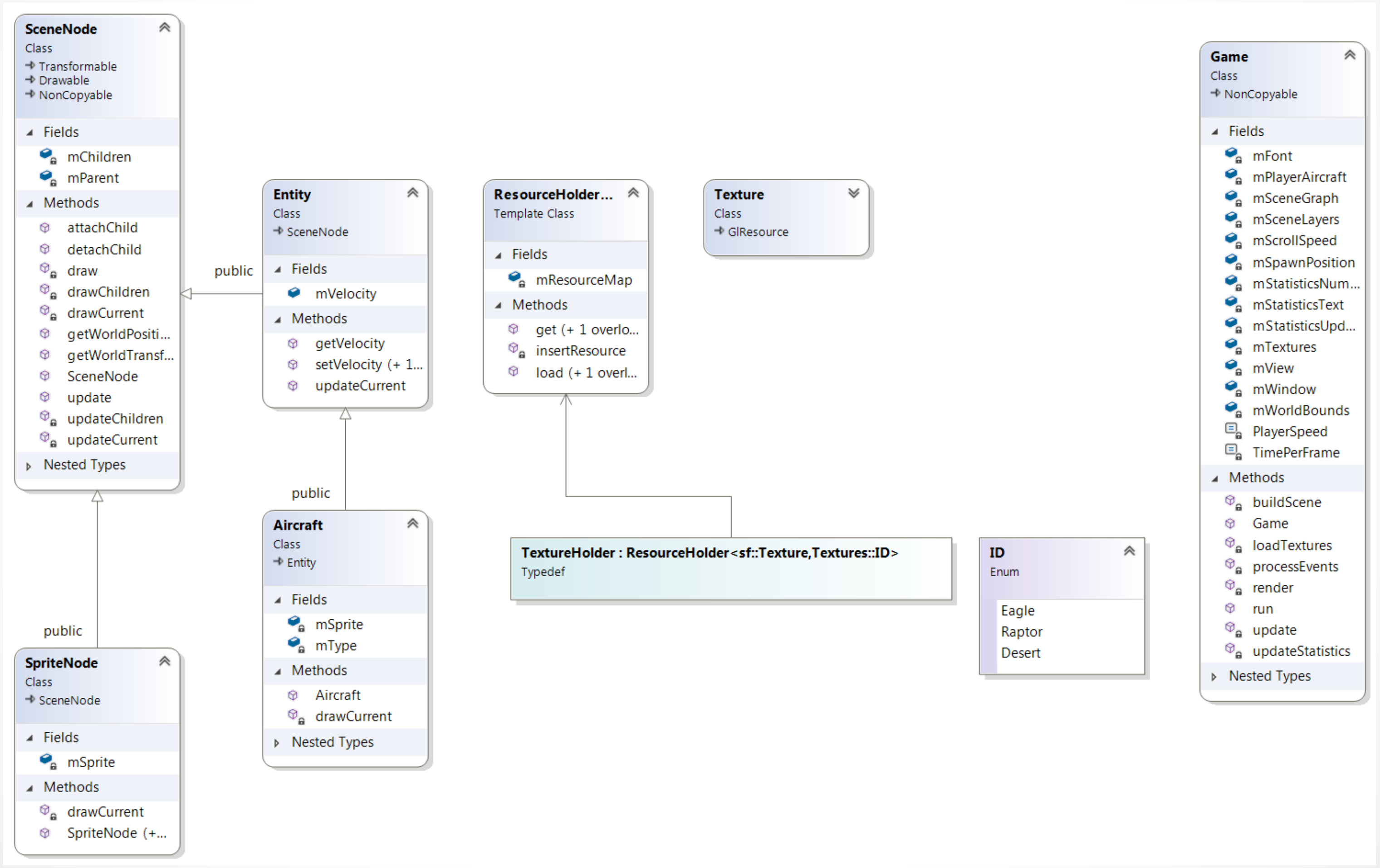
What's Left?

- Remember that the main() function is our entry point
- We can start to look at everything from there, including all the classes we've looked at
- The scene is built in the `World::buildScene()` method
- The update() and draw() methods of World encapsulate scene graph functionality
- The run() function in main gets everything going

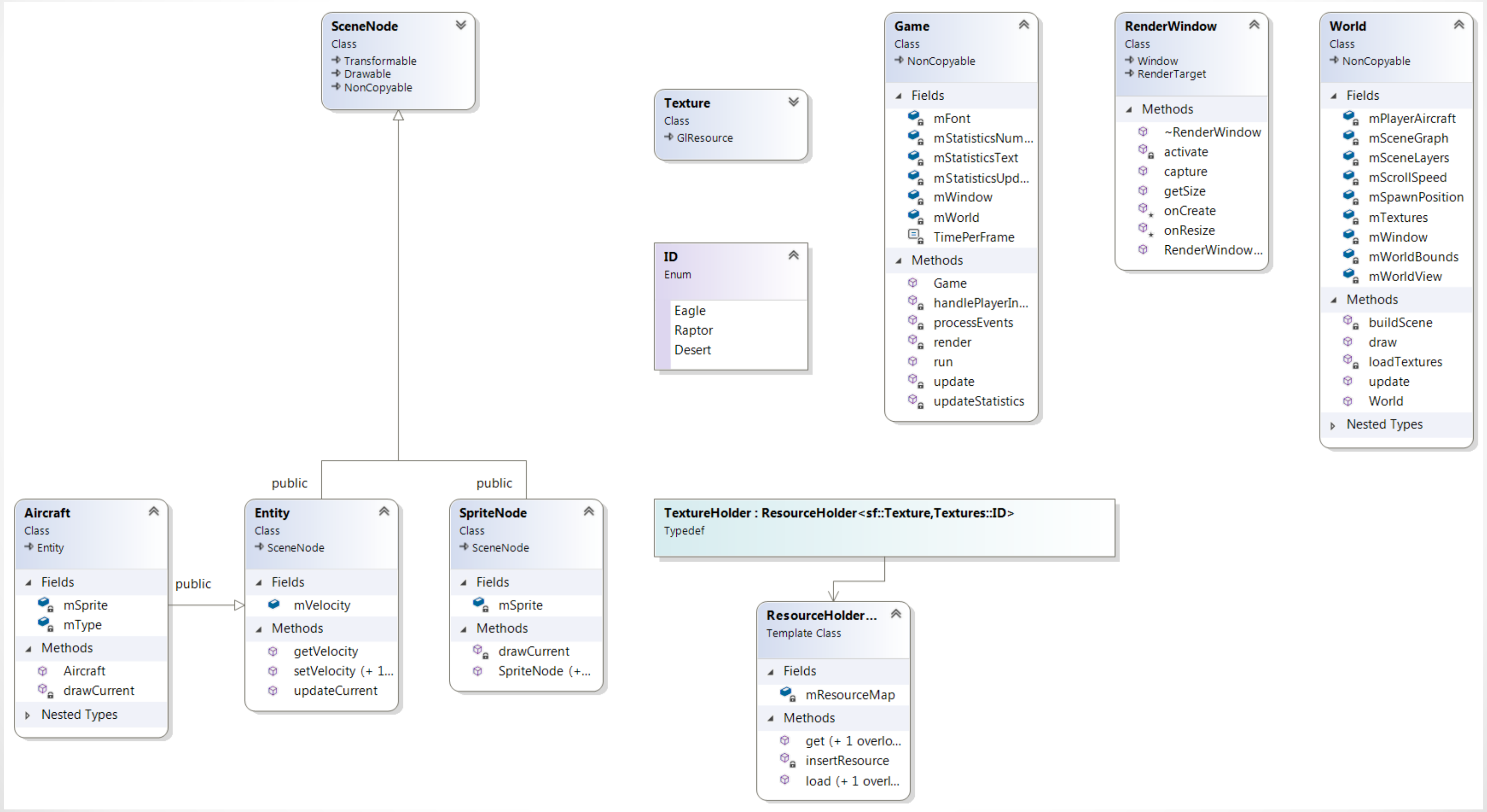
Entity


Entity
Class
Drawable
Transformable
Fields
mVelocity
Methods
getVelocity
setVelocity (+ 1...
update
public
Player2
Class
Entity
Fields
mSprite
texture2
Methods
draw
getSprite
Player2
update
Nested Types
ResourceHolder<Resource, Identifier>
Template Class
Fields
mResourceMap
Methods
get (+ 1 overload)
insertResource
load (+ 1 overload)
TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef
Game
Class
NonCopyable
Fields
airplane
landscape
mFont
mPlayer
mScrollSpeed
mSpawnPosition
mStatisticsNum...
mStatisticsText
mStatisticsUpd...
mTexture
mView
mWindow
mWorldBounds
player
player2
PlayerSpeed
textures
TimePerFrame
Methods
Game
processEvents
render
run
update
updateStatistics
Player
Class
Fields
mSprite
texture2
Methods
getSprite
Player
Nested Types
Type
Enum
Eagle
Raptor
ID
Enum
Landscape
Airplane
Texture
Class
GlResource

Scene Node


SceneNode
Class
Transformable
Drawable
NonCopyable
Fields
mChildren
mParent
Methods
attachChild
detachChild
draw
drawChildren
drawCurrent
getWorldPositi...
getWorldTransf...
SceneNode
update
updateChildren
updateCurrent
Nested Types
public
Entity
Class
SceneNode
Fields
mVelocity
Methods
getVelocity
setVelocity (+ 1...
updateCurrent
public
Aircraft
Class
Entity
Fields
mSprite
mType
Methods
Aircraft
drawCurrent
Nested Types
public
SpriteNode
Class
SceneNode
Fields
mSprite
Methods
drawCurrent
SpriteNode (+...
ResourceHolder...
Template Class
Fields
mResourceMap
Methods
get (+ 1 overlo...
insertResource
load (+ 1 overl...
Texture
Class
GlResource
TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef
ID
Enum
Eagle
Raptor
Desert
Game
Class
NonCopyable
Fields
mFont
mPlayerAircraft
mSceneGraph
mSceneLayers
mScrollSpeed
mSpawnPosition
mStatisticsNum...
mStatisticsText
mStatisticsUpd...
mTextures
mView
mWindow
mWorldBounds
PlayerSpeed
TimePerFrame
Methods
buildScene
Game
loadTextures
processEvents
render
run
update
updateStatistics
Nested Types

World


SceneNode
Class
Transformable
Drawable
NonCopyable
Texture
Class
GlResource
ID
Enum
Eagle
Raptor
Desert
Game
Class
NonCopyable
Fields
mFont
mStatisticsNum...
mStatisticsText
mStatisticsUpd...
mWindow
mWorld
TimePerFrame
Methods
Game
handlePlayerIn...
processEvents
render
run
update
updateStatistics
RenderWindow
Class
Window
RenderTarget
Methods
~RenderWindow
activate
capture
getSize
onCreate
onResize
RenderWindow...
World
Class
NonCopyable
Fields
mPlayerAircraft
mSceneGraph
mSceneLayers
mScrollSpeed
mSpawnPosition
mTextures
mWindow
mWorldBounds
mWorldView
Methods
buildScene
draw
loadTextures
update
World
Nested Types
public
public
Aircraft
Class
Entity
Fields
mSprite
mType
Methods
Aircraft
drawCurrent
Nested Types
public
Entity
Class
SceneNode
Fields
mVelocity
Methods
getVelocity
setVelocity (+ 1...
updateCurrent
SpriteNode
Class
SceneNode
Fields
mSprite
Methods
drawCurrent
SpriteNode (+...
TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef
ResourceHolder...
Template Class
Fields
mResourceMap
Methods
get (+ 1 overlo...
insertResource
load (+ 1 overl...