Game Engine Development II

Week 10

Hooman Salamat

AI
Projectiles and Collision

Objectives

- Examine enemies controlled by simple AI
- Implement projectiles
- Implement pickups that improve the player
- Examine collision detection and response
- Examine the world's update cycle and cleanup of entities

Introducing Hitpoints

- We add hitpoints to our Entity class:

```
class Entity : public SceneNode
{
public:
   explicit Entity(int hitpoints);
   void repair(int points);
   void damage(int points);
   void destroy();
   int getHitpoints() const;
   bool isDestroyed() const;
   ...
private:
   int mHitpoints;
   ...
};
```

Introducing Hitpoints (cont'd.)

- The repair(), damage() and destroy() methods of the Entity class all affect hitpoints in the expected way

AircraftData Structure

- To handle stats of all aircraft, we have a simple and handy structure:

```
struct AircraftData
{
  int hitpoints;
  float speed;
  Textures::ID texture;
};
```

AircraftData Structure

- With the struct in place, we define a function that initializes a table of data for the aircraft:

```
std::vector<AircraftData> initializeAircraftData()
{
   std::vector<AircraftData> data(Aircraft::TypeCount);
   data[Aircraft::Eagle].hitpoints = 100;
   data[Aircraft::Eagle].speed = 200.f;
   data[Aircraft::Eagle].texture = Textures::Eagle;
   data[Aircraft::Raptor].hitpoints = 20;
   data[Aircraft::Raptor].speed = 80.f;
   data[Aircraft::Raptor].texture = Textures::Raptor;
   ...
   return data;
}
```

AircraftData Structure

- With the struct in place, we define a function that initializes a table of data for the aircraft:

```
std::vector<AircraftData> initializeAircraftData()
{
   std::vector<AircraftData> data(Aircraft::TypeCount);
   data[Aircraft::Eagle].hitpoints = 100;
   data[Aircraft::Eagle].speed = 200.f;
   data[Aircraft::Eagle].texture = Textures::Eagle;
   data[Aircraft::Raptor].hitpoints = 20;
   data[Aircraft::Raptor].speed = 80.f;
   data[Aircraft::Raptor].texture = Textures::Raptor;
   ...
   return data;
}
```

AircraftData Structure (cont'd.)

- `initializeAircraftData()` is declared in `DataTables.hpp` and defined in `DataTables.cpp`
- In order to avoid name collisions in other files, we use an anonymous namespace

```
namespace
{
   const std::vector<AircraftData> Table = initializeAircraftData();
}
```

Displaying Text

- We create a `TextNode` class which inherits `SceneNode` as shown in the following:

```
class TextNode : public SceneNode
{
public:
   explicit TextNode(const FontHolder& fonts,
   const std::string& text);
   void setString(const std::string& text);
private:
   virtual void drawCurrent(sf::RenderTarget& target,
   sf::RenderStates states) const;
private:
   sf::Text mText;
};
```

Displaying Text (cont'd.)

```
TextNode::TextNode(const FontHolder& fonts, const std::string& text)
{
    mText.setFont(fonts.get(Fonts::Main));
    mText.setCharacterSize(20);
    setString(text);
}

void TextNode::drawCurrent(sf::RenderTarget& target, sf::RenderStates
states) const
{
    target.draw(mText, states);
}

void TextNode::setString(const std::string& text)
{
    mText.setString(text);
    centerOrigin(mText);
}
```

Aircraft Changes

- In the Aircraft constructor:
  - We create a text node and attach it to the aircraft itself
  - We keep a pointer `mHealthDisplay` as a member variable

```
std::unique_ptr<TextNode> healthDisplay(new TextNode(fonts, ""));
mHealthDisplay = healthDisplay.get();
attachChild(std::move(healthDisplay));
```

- In the Aircraft update:

```
mHealthDisplay->setString(toString(getHitpoints()) + " HP");
mHealthDisplay->setPosition(0.f, 50.f);
mHealthDisplay->setRotation(-getRotation());
```

Movement

- Enemies are instances of the `Aircraft` class
- Their behavior is simple:
  - Appear at the top of the screen and move down
  - We've defined a Direction struct that has an angle and distance:

```
struct Direction
{
   Direction(float angle, float distance);
   float angle;
   float distance;
};
```

Movement (cont'd.)

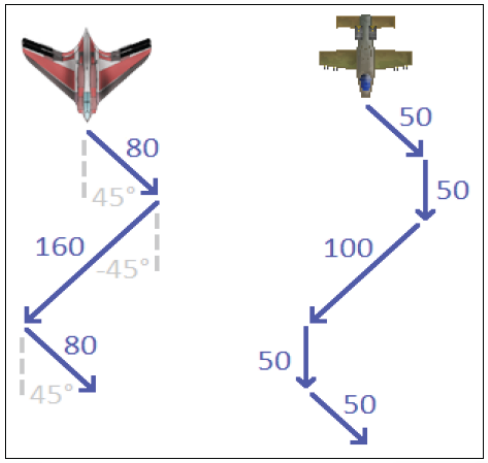
```
struct AircraftData
{
   int hitpoints;
   float speed;
   Textures::ID texture;
   std::vector<Direction> directions;
};

data[Aircraft::Raptor].directions.push_back(Direction( 45, 80));
data[Aircraft::Raptor].directions.push_back(Direction(-45, 160));
data[Aircraft::Raptor].directions.push_back(Direction( 45, 80));
```

Adding a simple AI to handle Movement

- The Avenger airplane also includes diagonal movement:

```
data[Aircraft::Avenger].directions.push_back(Direction(+45, 50));
data[Aircraft::Avenger].directions.push_back(Direction( 0, 50));
data[Aircraft::Avenger].directions.push_back(Direction(-45, 100));
data[Aircraft::Avenger].directions.push_back(Direction( 0, 50));
data[Aircraft::Avenger].directions.push_back(Direction(+45, 50));
```



Movement (cont'd.)

```
void Aircraft::updateMovementPattern(sf::Time dt)
{
    const std::vector<Direction>& directions = Table[mType].directions;
    if (!directions.empty())
    {
        float distanceToTravel = directions[mDirectionIndex].distance;
        if (mTravelledDistance > distanceToTravel)
        {
          mDirectionIndex = (mDirectionIndex + 1) % directions.size();
          mTravelledDistance = 0.f;
        }
        float radians = toRadian(directions[mDirectionIndex].angle + 90.f);
        float vx = getMaxSpeed() * std::cos(radians);
        float vy = getMaxSpeed() * std::sin(radians);
        setVelocity(vx, vy);
        mTravelledDistance += getMaxSpeed() * dt.asSeconds();
    }
}
```

Enemy Spawning

```
struct SpawnPoint
{
   SpawnPoint(Aircraft::Type type, float x, float y);
   Aircraft::Type type;
   float x;
   float y;
};
```

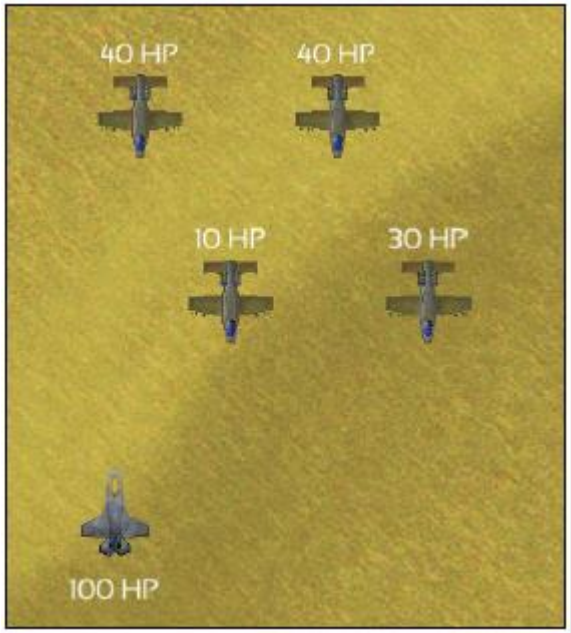
- A member variable `World::mEnemySpawnPoints` of type `std::vector<SpawnPoint>` holds all future spawn points

Enemy Spawning (cont'd.)

```
void World::spawnEnemies()
{
    while (!mEnemySpawnPoints.empty() && mEnemySpawnPoints.back().y
    > getBattlefieldBounds().top)
    {
        SpawnPoint spawn = mEnemySpawnPoints.back();
        std::unique_ptr<Aircraft> enemy( new Aircraft(spawn.type,
          mTextures, mFonts));
        enemy->setPosition(spawn.x, spawn.y);
        enemy->setRotation(180.f);
        mSceneLayers[Air]->attachChild(std::move(enemy));
        mEnemySpawnPoints.pop_back();
    }
}
```

Enemy Spawning (cont'd.)

```
void World::addEnemies()
{
	addEnemy(Aircraft::Raptor, 0.f, 500.f);
	addEnemy(Aircraft::Avenger, -70.f, 1400.f);
	...
	std::sort(mEnemySpawnPoints.begin(), mEnemySpawnPoints.end(),
	[] (SpawnPoint lhs, SpawnPoint rhs)
	{
		return lhs.y < rhs.y;
	});
}
```



Projectiles

```
class Projectile : public Entity
{
public:
    enum Type
    {
       AlliedBullet,
       EnemyBullet,
       Missile,
       TypeCount
    };
public:
    Projectile(Type type,
    const TextureHolder& textures);
    void guideTowards(sf::Vector2f position);
    bool isGuided() const;
    virtual unsigned int getCategory() const;
    virtual sf::FloatRect getBoundingRect() const;
    float getMaxSpeed() const;
    int getDamage() const;
```

Projectiles (cont'd.)

```
private:
    virtual void updateCurrent(sf::Time dt, CommandQueue& commands);
    virtual void drawCurrent(sf::RenderTarget& target, sf::RenderStates states) const;

private:
    Type mType;
    sf::Sprite mSprite;
    sf::Vector2f mTargetDirection;
};
```

- Constructor:

```
Projectile::Projectile(Type type, const TextureHolder& textures) : Entity(1)
    , mType(type), mSprite(textures.get(Table[type].texture))
{
    centerOrigin(mSprite);
}
```

Firing Projectiles

```
Player::Player()
{
    // Set initial key bindings
    mKeyBinding[sf::Keyboard::Left] = MoveLeft;
    mKeyBinding[sf::Keyboard::Right] = MoveRight;
    mKeyBinding[sf::Keyboard::Up] = MoveUp;
    mKeyBinding[sf::Keyboard::Down] = MoveDown;
    mKeyBinding[sf::Keyboard::Space] = Fire;
    mKeyBinding[sf::Keyboard::M] = LaunchMissile;
    // ...
}

void Player::initializeActions()
{
    // ...
    mActionBinding[Fire].action = derivedAction<Aircraft>(
          std::bind(&Aircraft::fire, _1));
    mActionBinding[LaunchMissile].action =derivedAction<Aircraft>(
          std::bind(&Aircraft::launchMissile, _1));
}
```

Firing Projectiles (cont'd.)

```
void Aircraft::checkProjectileLaunch(sf::Time dt, CommandQueue&
commands)
{
   if (mIsFiring && mFireCountdown <= sf::Time::Zero)
    {
       commands.push(mFireCommand);
       mFireCountdown += sf::seconds(1.f / (mFireRateLevel+1));
       mIsFiring = false;
    }
    else if (mFireCountdown > sf::Time::Zero)
    {
       mFireCountdown -= dt;
    }
    if (mIsLaunchingMissile)
    {
       commands.push(mMissileCommand);
       mIsLaunchingMissile = false;
    }
}
```

Firing Projectiles (cont'd.)

```
bool Player::isRealtimeAction(Action action)
{
    switch (action)
    {
        case MoveLeft:
        case MoveRight:
        case MoveDown:
        case MoveUp:
        case Fire:
          return true;
        default:
          return false;
    }
}
```

Firing Projectiles (cont'd.)

```
Aircraft::Aircraft(Type type, const TextureHolder& textures)
{
   mFireCommand.category = Category::SceneAirLayer;
   mFireCommand.action = [this, &textures] (SceneNode& node,
     sf::Time)
   {
        createBullets(node, textures);
   };

    mMissileCommand.category = Category::SceneAirLayer;
    mMissileCommand.action = [this, &textures] (SceneNode& node,
      sf::Time)
    {
      createProjectile(node, Projectile::Missile, 0.f, 0.5f,
      textures);
    };
}
```

Firing Projectiles (cont'd.)

```
void Aircraft::createBullets(SceneNode& node, const TextureHolder& textures) const
{
    Projectile::Type type = isAllied() ? Projectile::AlliedBullet :
       Projectile::EnemyBullet;
    switch (mSpreadLevel)
    {
       case 1:
         createProjectile(node, type, 0.0f, 0.5f, textures);
         break;
       case 2:
         createProjectile(node, type, -0.33f, 0.33f, textures);
         createProjectile(node, type, +0.33f, 0.33f, textures);
         break;
       case 3:
         createProjectile(node, type, -0.5f, 0.33f, textures);
         createProjectile(node, type, 0.0f, 0.5f, textures);
         createProjectile(node, type, +0.5f, 0.33f, textures);
         break;
    }
}
```

Firing Projectiles (cont'd.)

```
void Aircraft::createProjectile(SceneNode& node,
  Projectile::Type type, float xOffset, float yOffset,
  const TextureHolder& textures) const
{
    std::unique_ptr<Projectile> projectile(
      new Projectile(type, textures));
    sf::Vector2f offset(
      xOffset * mSprite.getGlobalBounds().width,
      yOffset * mSprite.getGlobalBounds().height);
    sf::Vector2f velocity(0, projectile->getMaxSpeed());

    float sign = isAllied() ? -1.f : +1.f;
    projectile->setPosition(getWorldPosition() + offset * sign);
    projectile->setVelocity(velocity * sign);
    node.attachChild(std::move(projectile));
}
```

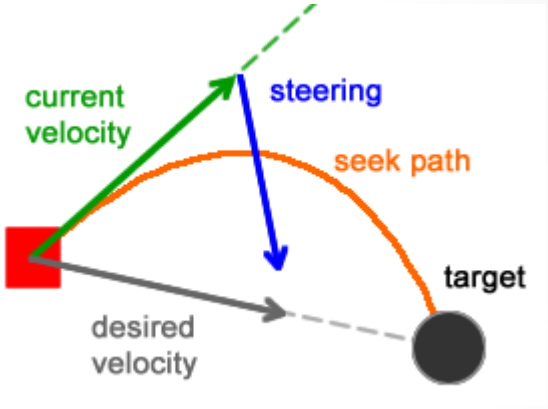
Homing Missiles

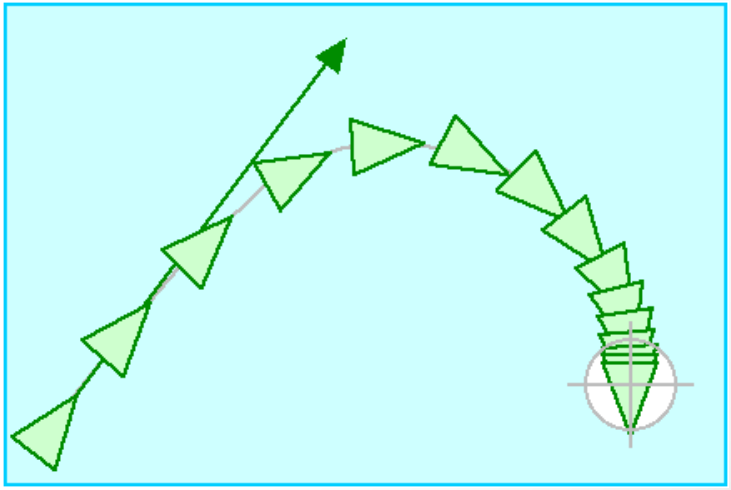
```
bool Projectile::isGuided() const
{
   return mType == Missile;
}

void Projectile::guideTowards(sf::Vector2f position)
{
   assert(isGuided());
   mTargetDirection = unitVector(position - getWorldPosition());
}
```

AI: What is the seeking behavior?

- The seeking behavior is the idea that an AI agent "seeks" to have a certain velocity (vector).
- To start, we'll need to have 2 things:
  - An initial velocity *(a vector)*
  - A desired velocity *(also a vector)*
- First, we need to find the velocity needed for our agent to reach a desired point...
  - This is usually a subtraction of the current position of the agent and the desired position.
- Secondly, we must also find the agent's current velocity, which is usually already available in most game engines.
- Next, we need to find the vector difference between the desired velocity and the agent's current velocity.
  - it literally gives us a vector that gives the desired velocity when we add it to that agent's current velocity. We will call it "steering velocity".





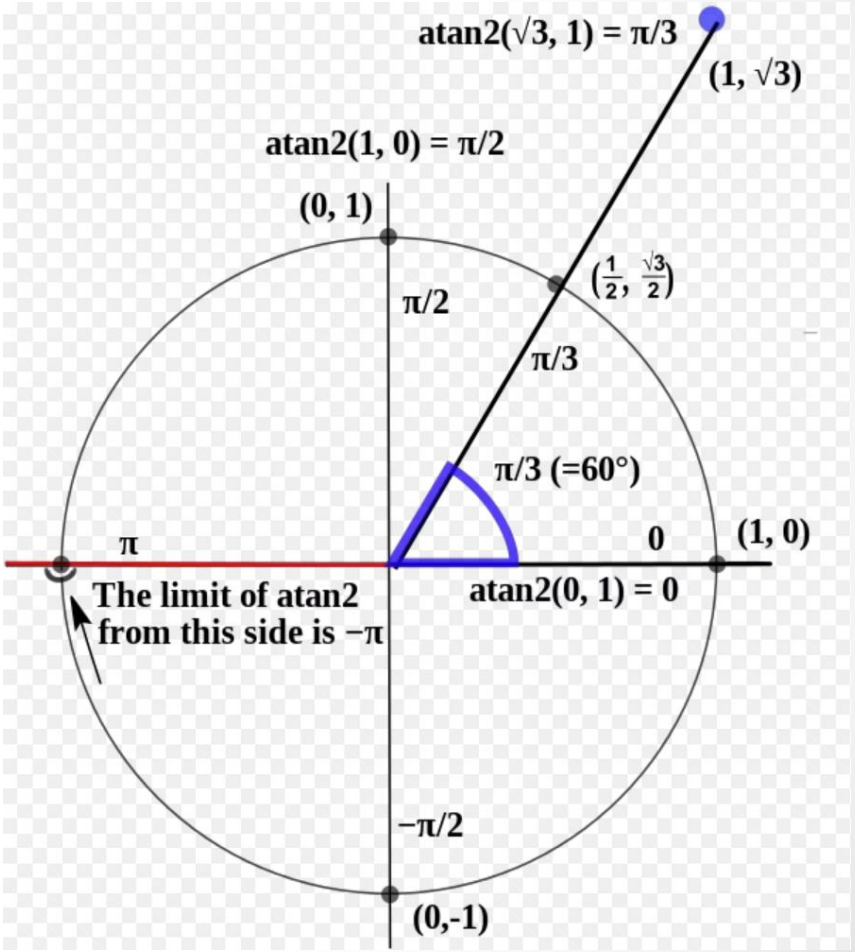
atan2

```
#include <iostream>
#include <cmath>

int main()
{
    // normal usage: the signs of the two arguments determine the quadrant
    std::cout << "(x:+1,y:+1) cartesian is (r:" << hypot(1,1)
              << ",phi:" << atan2(1,1) << ") polar\n"  // atan2(1,1) = +pi/4, Quad I
              << "(x:-1,y:+1) cartesian is (r:" << hypot(1,-1)
              << ",phi:" << atan2(1,-1) << ") polar\n" // atan2(1, -1) = +3pi/4, Quad II
              << "(x:-1,y:-1) cartesian is (r:" << hypot(-1,-1)
              << ",phi:" << atan2(-1,-1) << ") polar\n" // atan2(-1,-1) = -3pi/4, Quad III
              << "(x:+1,y:-1) cartesian is (r:" << hypot(-1,1)
              << ",phi:" << atan2(-1,1) << ") polar\n"; // atan2(-1, 1) = -pi/4, Quad IV
    // special values
    std::cout << "atan2(0, 0) = " << atan2(0,0)
              << " atan2(0,-0) = " << atan2(0,-0.0) << '\n'
              << "atan2(7, 0) = " << atan2(7,0)
              << " atan2(7,-0) = " << atan2(7,-0.0) << '\n';
}
```

Std:atan2

- Computes the arc tangent of y/x using the signs of arguments to determine the correct quadrant.



Homing Missiles (cont'd.)

```
void Projectile::updateCurrent(sf::Time dt,
  CommandQueue& commands)
{
   if (isGuided())
    {
        const float approachRate = 200.f;
        sf::Vector2f newVelocity = unitVector(approachRate
          * dt.asSeconds() * mTargetDirection + getVelocity());
        newVelocity *= getMaxSpeed();
        float angle = std::atan2(newVelocity.y, newVelocity.x);
        setRotation(toDegree(angle) + 90.f);
        setVelocity(newVelocity);
    }
    Entity::updateCurrent(dt, commands);
}
```

Homing Missiles (cont'd.)

```
void World::guideMissiles()
{
    Command enemyCollector;
    enemyCollector.category = Category::EnemyAircraft;
    enemyCollector.action = derivedAction<Aircraft>(
      [this] (Aircraft& enemy, sf::Time)
    {
    if (!enemy.isDestroyed())
       mActiveEnemies.push_back(&enemy);
   });

   Command missileGuider;
   missileGuider.category = Category::AlliedProjectile;
   missileGuider.action = derivedAction<Projectile>(
   [this] (Projectile& missile, sf::Time)
   {
        // Ignore unguided bullets
        if (!missile.isGuided())
                return;
```

Homing Missiles (cont'd.)

```
        float minDistance = std::numeric_limits<float>::max();
        Aircraft* closestEnemy = nullptr;
        FOREACH(Aircraft* enemy, mActiveEnemies)
        {
          float enemyDistance = distance(missile, *enemy);
          if (enemyDistance < minDistance)
          {
                closestEnemy = enemy;
                minDistance = enemyDistance;
          }
        }
        if (closestEnemy)
          missile.guideTowards(       closestEnemy-
          >getWorldPosition());
});
```

Pickups

```
class Pickup : public Entity
{
public:
    enum Type
    {
        HealthRefill,
        MissileRefill,
        FireSpread,
        FireRate,
        TypeCount
    };
public:
     Pickup(Type type, const TextureHolder& textures);
     virtual unsigned int getCategory() const;
     virtual sf::FloatRect getBoundingRect() const;
     void apply(Aircraft& player) const;
protected:
     virtual void drawCurrent(sf::RenderTarget& target, sf::RenderStates states) const;
private:
     Type mType;
     sf::Sprite mSprite;
};
```

Collisions

- Collisions for the game will deal with bounding rectangles
- Below in an example where `getWorldTransform()` multiplies `sf::Transform` objects from the scene root to the leaf
  - sf::Transform::transformRect() transforms a rectangle
    - May enlarge it if there is a rotation
  - sf::Sprite::getGlobalBounds() returns the sprite's bounding rectangle
    - Relative to the aircraft

```
sf::FloatRect Aircraft::getBoundingRect() const
{
    return
    getWorldTransform().transformRect(mSprite.getGlobalBounds());
}
```

Finding Collision Pairs

```
void SceneNode::checkNodeCollision(SceneNode& node, std::set<Pair>&
collisionPairs)
{
     if (this != &node && collision(*this, node) && !isDestroyed() && !node.isDestroyed())
        collisionPairs.insert(std::minmax(this, &node));
     FOREACH(Ptr& child, mChildren)
        child->checkNodeCollision(node, collisionPairs);
}

void SceneNode::checkSceneCollision(SceneNode& sceneGraph, std::set<Pair>& collisionPairs)
{
    checkNodeCollision(sceneGraph, collisionPairs);
    FOREACH(Ptr& child, sceneGraph.mChildren)
          checkSceneCollision(*child, collisionPairs);
}
```

Reacting to Collisions

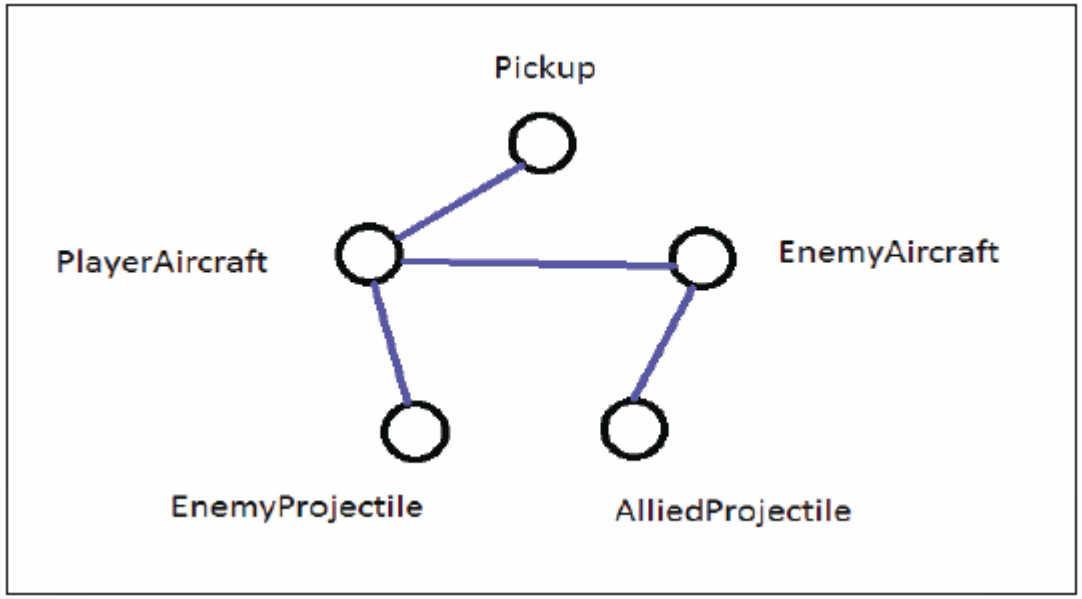
```
bool matchesCategories(SceneNode::Pair& colliders,
  Category::Type type1, Category::Type type2)
{
    unsigned int category1 = colliders.first->getCategory();
    unsigned int category2 = colliders.second->getCategory();
    if (type1 & category1 && type2 & category2)
    {
       return true;
    }
    else if (type1 & category2 && type2 & category1)
    {
       std::swap(colliders.first, colliders.second);
       return true;
    }
    else
    {
       return false;
    }
}
```

Reacting to Collisions

```
void World::handleCollisions()
{
    std::set<SceneNode::Pair> collisionPairs;
    mSceneGraph.checkSceneCollision(mSceneGraph, collisionPairs);
    FOREACH(SceneNode::Pair pair, collisionPairs)
    {
        if (matchesCategories(pair,
        Category::PlayerAircraft, Category::EnemyAircraft))
        {
          ... // React to player-enemy collision
        }
    }
}
```

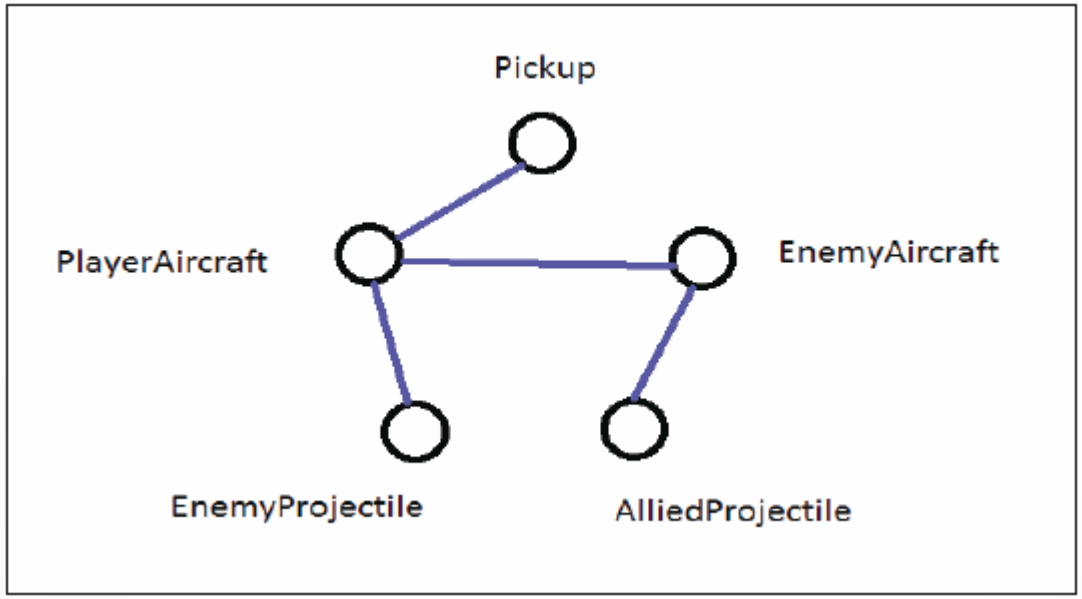
Reacting to Collisions

- We have four combinations of categories which trigger a collision, as shown in the following diagram:



Reacting to Collisions

- We have four combinations of categories which trigger a collision, as shown in the following diagram:



Reacting to Collisions

- We need four calls to matchesCategories() in order to react to all possible combinations
  - Code goes in the 'React to player-enemy collision' part of previous code

```
if (matchesCategories(pair, Category::PlayerAircraft, Category::EnemyAircraft))
{
    auto& player = static_cast<Aircraft&>(*pair.first);
    auto& enemy = static_cast<Aircraft&>(*pair.second);
    player.damage(enemy.getHitpoints());
    enemy.destroy();
}

else if (matchesCategories(pair, Category::PlayerAircraft, Category::Pickup))
{
    auto& player = static_cast<Aircraft&>(*pair.first);
    auto& pickup = static_cast<Pickup&>(*pair.second);
    pickup.apply(player);
    pickup.destroy();
}
```

Reacting to Collisions

```
else if (matchesCategories(pair, Category::EnemyAircraft, Category::AlliedProjectile)
    || matchesCategories(pair,
Category::PlayerAircraft, Category::EnemyProjectile))
{
    auto& aircraft = static_cast<Aircraft&>(*pair.first);
    auto& projectile = static_cast<Projectile&>(*pair.second);
    aircraft.damage(projectile.getDamage());
    projectile.destroy();
}
```

Cleaning Up

```
bool Entity::isDestroyed() const
{
	return mHitpoints <= 0;
}

bool SceneNode::isMarkedForRemoval() const
{
	return isDestroyed();
}

bool Aircraft::isMarkedForRemoval() const
{
	return mIsMarkedForRemoval;
}
void SceneNode::removeWrecks()
{
	auto wreckfieldBegin = std::remove_if(mChildren.begin(),
	mChildren.end(), std::mem_fn(&SceneNode::isMarkedForRemoval));
	mChildren.erase(wreckfieldBegin, mChildren.end());
	std::for_each(mChildren.begin(), mChildren.end(),
	std::mem_fn(&SceneNode::removeWrecks));
}
```

Cleaning Up (cont'd.)

```
void World::destroyEntitiesOutsideView()
{
    Command command;
    command.category = Category::Projectile | Category::EnemyAircraft;
    command.action = derivedAction<Entity>([this] (Entity& e, sf::Time)
    {
       if (!getBattlefieldBounds().intersects(e.getBoundingRect()))
         e.destroy();
    });

    mCommandQueue.push(command);
}
```

Final Update

```
void World::update(sf::Time dt)
{
    mWorldView.move(0.f, mScrollSpeed * dt.asSeconds());
    mPlayerAircraft->setVelocity(0.f, 0.f);

    destroyEntitiesOutsideView();
    guideMissiles();

    while (!mCommandQueue.isEmpty())
       mSceneGraph.onCommand(mCommandQueue.pop(), dt);
    adaptPlayerVelocity();
    handleCollisions();
    mSceneGraph.removeWrecks();
    spawnEnemies();

    mSceneGraph.update(dt, mCommandQueue);
    adaptPlayerPosition();
}
```