Cocos2d-x Game Development Blueprints

Build a plethora of games for various genres using one of the most powerful game engines, Cocos2d-x

Karan Sequeira
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Back to the Drawing Board'
- A synopsis of the book’s content
- More information on Cocos2d-x Game Development Blueprints
About the Author

Karan Sequeira is a budding game developer with 4 years of experience in the mobile game development industry. He started out as a JavaScript programmer, developing HTML5 games, and fell in love with C++ when he moved to Cocos2d-x. He has a handful of games on the iOS and Android app stores and is currently working for an organization in the in-flight entertainment industry. He is extremely passionate about learning new technologies and dreams about building an artificially intelligent super computer that can fall in love.
Preface

Cocos2d-x offers you, as a game developer, an open source engine that enables you to effortlessly build powerful 2D games and publish them on a multitude of popular platforms. In today's fast-paced world, what more could a game developer need?

This book gets you started by introducing the continually evolving HTML5 platform and familiarizes you with the Cocos2d-html5 framework.

This book is written to get you well versed with the many versatile features of Cocos2d-x, by demonstrating the creation of nine different games—from arcade style games to new age physics games and from simple puzzle games to strategic tower defense games! All the while, focusing on gameplay and letting Cocos2d-x do all the heavy lifting for you.

For the finale, you will learn how to build your games for Android and Windows Phone 8, bringing you to a full circle on Cocos2d-x.

What this book covers

Chapter 1, A Colorful Start, introduces the HTML5 version of the Cocos2d family by creating a simple and colorful puzzle game. You will learn how to set up your environment to develop using the Cocos2d-html5 engine. You also get familiar with the various actions you can use to get things done in an instant.

Chapter 2, How to Fly a Dragon!, concludes your HTML5 journey and will show you how to use sprite sheets, how to implement simple gravity, and use rectangular collision detection.

Chapter 3, Not Just a Space Game, shows you how to create a multiplatform Cocos2d-x project. You will also learn to create a level-based arcade game where you learn how to parse XML, play particle effects, and extend the engine by inheriting from its primary classes.
Preface

Chapter 4, *Back to the Drawing Board*, shows you just how easy it is to make a game that completely uses primitive drawing for its entities. You will also learn how to implement simple accelerometer controls and how to add difficulty progression to an endless game.

Chapter 5, *Let's Get Physical!*, guides you through the process of using Box2D to build a physics game. You will also learn about state machines and object pools.

Chapter 6, *Creativity with Textures*, shows you how to fire OpenGL commands and render your own sprite. You will also learn to use Box2D to make a picturesque side-scroller game.

Chapter 7, *Old is Gold!*, introduces you to the Tiled map editor and the basics of a tile-based game. You will also learn how to implement a reusable tile-based collision detection algorithm.

Chapter 8, *Box2D Meets RUBE*, introduces you to RUBE, which is a powerful physics editor. You will use it to create levels for a slingshot type physics game. You will also learn about ray casting and how to implement explosions in Box2D.

Chapter 9, *The Two Towers*, shows you how easy it is to implement a tower defense game in Cocos2d-x. In the process, you will learn how to implement scalable architectures for your enemies and towers. You will also learn how a simple gesture recognizer works and how you can control the speed of your gameplay.

Chapter 10, *Cross-platform Building*, shows you how to set up your environment and build one of your games on Android and Windows Phone 8.
Back to the Drawing Board

In our fourth game, we head back to the drawing board quite literally. We will create each element of the game using primitive drawing. Though that may seem a bit cumbersome to accomplish, Cocos2d-x makes it simple. Finally, we will add tilt controls to make this game even cooler!

In this chapter, you'll learn:

- How to use and extend the CCDrawNode class
- How to implement tilt controls
- How to add time-based difficulty progression

An introduction to Inverse Universe

This is yet another game set in space where you're surrounded by dark creatures that can destroy you by merely making contact. What makes matters worse is that you are absolutely powerless. The only way you can actually kill enemies is by the virtue of three wondrous power-ups: the bomb, the missile launcher, and the glorious shield.

The controls for this game are based on the accelerometer. Users will have to tilt their device to navigate the player ship. Inverse Universe is not about winning or losing; it is about survival. Thus, we won't be designing levels for this game like we did in the previous chapter. Instead, we will design difficulty levels in such a way that the game will get progressively more difficult as time goes by. This also lets the game stay open-ended and variable.
This is what you will have accomplished at the end of this chapter:

![Game screenshot](image)

**The CCDrawNode class**

There are two ways to draw primitive shapes using the functionality provided by Cocos2d-x: one is using the various functions in CCDrawingPrimitives and the other way is using the CCDrawNode class. While the functions defined in CCDrawingPrimitives offer abundant variety, they also incur a performance penalty. The CCDrawNode class, on the other hand, provides just three different drawing functions and batches all its draw calls together to give much better performance. It also inherits from CNode and thus we can run a few common actions on it! So, we will be using CCDrawNode to render our primitive shapes for this game.

Everything you saw in the preceding screenshot has been drawn using code—except for the score of course. All the main entities of our game, that is, the player, enemies, and power-ups, will inherit from CCDrawNode as you will see shortly. If you're confused about how primitive drawing can yield the preceding results, the following section will make things clearer.
Figuring out the geometry

The CCDrawNode class provides just three functions. These functions are drawDot, drawSegment, and drawPolygon, which enable you to draw a color-filled circle, a color-filled segment, and a color-filled polygon with a separate border, respectively. We will write a simple yet extremely resourceful function to help us with the generation of vertices for our game elements. However, before we write this utilitarian function, let’s take a closer look at each element and the vertices they contain:

<table>
<thead>
<tr>
<th>Game element</th>
<th>Visual</th>
<th>Visual with vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td><img src="image1.png" alt="Visual" /></td>
<td><img src="image2.png" alt="Visual with vertices" /></td>
</tr>
<tr>
<td>Enemy</td>
<td><img src="image3.png" alt="Visual" /></td>
<td><img src="image4.png" alt="Visual with vertices" /></td>
</tr>
<tr>
<td>Shield</td>
<td><img src="image5.png" alt="Visual" /></td>
<td><img src="image6.png" alt="Visual with vertices" /></td>
</tr>
<tr>
<td>Bomb</td>
<td><img src="image7.png" alt="Visual" /></td>
<td><img src="image8.png" alt="Visual with vertices" /></td>
</tr>
<tr>
<td>Missile launcher</td>
<td><img src="image9.png" alt="Visual" /></td>
<td><img src="image10.png" alt="Visual with vertices" /></td>
</tr>
</tbody>
</table>
I have highlighted the vertices on each of these shapes to underline a basic geometrical object that we will use to our advantage in this game—the regular polygon. As most of you may know, a regular polygon is a polygon that is equiangular and equilateral. The interesting thing about a regular polygon is that we can use the parametric equation of a circle to get its vertices. This is exactly what the `GetRegularPolygonVertices` function from our helper class `GameGlobals` does. Let's take a look:

```cpp
void GameGlobals::GetRegularPolygonVertices(vector<CCPoint> &vertices, int num_vertices, float circum_radius, float start_angle)
{
    vertices.clear();
    float delta_theta = 2 * M_PI / num_vertices;
    float theta = start_angle;
    for(int i = 0; i < num_vertices; ++i, theta += delta_theta)
    {
        vertices.push_back(CCPoint(circum_radius * cosf(theta),
                                    circum_radius * sinf(theta)));
    }
}
```

We passed in a `vector` of `CCPoint` that will actually hold the vertices along with the number of vertices, the radius of the encompassing circle, and the angle the first vertex will make with the x axis. The function then calculates the difference in angles between successive vertices of the polygon based on the number of vertices the polygon is supposed to have. Then, initializing `theta` with `start_angle`, we run a loop to generate each vertex using the parametric equation of the circle. To summarize, let's take a look at the output of this function:

```cpp
GameGlobals::GetRegularPolygonVertices(vertices, 3, 50, 0)
```

This generates the vertices of the following triangle:
While `GameGlobals::GetRegularPolygonVertices(vertices, 3, 50, M_PI/2)` generates the vertices of the following triangle:

![Triangle](image)

So, we have everything we need to draw most of the elements in our game. You can already fill in the blanks and see that the Bomb, which happens to be a power-up, can be summed up as a brown circle containing a green triangle. Similarly, the Shield power-up is a brown circle containing a cyan hexagon, but MissileLauncher and Enemy might not be that straightforward. We shall cross that bridge when we get there, but now we need to focus on the main elements of the game world.

## Defining the elements of Inverse Universe

In this section, we will discuss the behavior of each element of the game in as much detail as possible, starting from the player, to the enemies and to each of the three power-ups. So, let's begin with the Player class.

### The Player class

The Player class will inherit from `CCDrawNode` and will have a radius to be used for circular collision detection, a speed variable that will be set based on the input from the accelerometer, and a reference to `GameWorld`. Let's begin by defining the `init` function for the Player class:

```cpp
bool Player::init()
{
    if(!CCDrawNode::init())
        return false;

    // generate vertices for the player
    CCPoint vertices[] = {
        CCPoint(PLAYER_RADIUS * 1.75f, 0),
        CCPoint(PLAYER_RADIUS * -0.875f, PLAYER_RADIUS),
        CCPoint(PLAYER_RADIUS * -1.75, 0),
        CCPoint(PLAYER_RADIUS * -0.875f, PLAYER_RADIUS * -1),
        CCPoint(PLAYER_RADIUS * 1.75f, 0),
        CCPoint(PLAYER_RADIUS * -0.875f, PLAYER_RADIUS * -1)};

    // draw a green coloured player
```
drawPolygon(vertices, 4, ccc4f(0, 0, 0, 0), 1.5f, ccc4f(0, 1, 0, 1));

    scheduleUpdate();
    return true;
}

The init function for the parent class is called first before defining the vertices that will make the player's ship. The variable PLAYER_RADIUS used to calculate the vertices has been defined in the GameGlobals.h file and is equivalent to 20 pixels. We then call the drawPolygon function, passing in the array of vertices, number of vertices, fill color, border thickness, and border color. We need all this to draw the player. Since this is a test of survival, our player must dodge enemies to prevent being destroyed but invariably death catches up to us all. The player dies in the following way in the Die function:

    void Player::Die()
    {
        // don't die if already dying
        if(is_dying_)
            return;
        is_dying_ = true;
        // stop moving
        speed_.x = 0;
        speed_.y = 0;
        float shake_duration = 0.5f;
        int num_shakes = 8;
        // create & animate the death and end the game afterwards
        CCActionInterval* shake = CCSpawn::createWithTwoActions(
            CCScaleTo::create(shake_duration, 1.2f), CCRepeat::create(
                CCSequence::createWithTwoActions(CCRotateBy::create(shake_ 
                    duration/(num_shakes*2), -20.0), CCRotateBy::create(shake_ 
                    duration/(num_shakes*2), 20.0)), num_shakes));
        CCActionInterval* shrink = CCEaseSineIn::create(CCScaleTo::create(0 
            .1f, 0.0f));
        CCActionInterval* death = CCSequence::create(shake, shrink, NULL);
        runAction(CCSequence::createWithTwoActions(death, 
            CCCallFunc::create(this, 
                callfunc_selector(Player::Dead))));
        SOUND_ENGINE->playEffect("blast_player.wav");
    }

    void Player::Dead()
    {
        game_world_->GameOver();
    }
A collision may occur in every tick of the game loop thus we prevent the Die function from being called repeatedly by checking and immediately enabling the is_dying_flag. We then stop movement, animate the death with the help of a few composite actions, and also play a sound. The callback to the function Dead tells GameWorld to finish the game. There is more to the Player class than just this, but we will define the remaining behavior, such as the movement, when we get to the Moving the player section of this chapter.

The Enemy class

The enemies in this game are deadly creatures that follow the player throughout the game and get faster the longer they live. Let's give them their lethal characteristics by first defining their init function:

```cpp
bool Enemy::init(GameWorld* instance)
{
    if(!CCDrawNode::init())
        return false;

    game_world_ = instance;
    CCPoint vertices[NUM_SPIKES*2];
    GenerateVertices(vertices);

    // draw the star shaped polygon filled with red colour
    drawPolygon(vertices, NUM_SPIKES*2, ccc4f(1, 0, 0, 1), 1.5f,
                ccc4f(1, 0, 0, 1));
    // draw a black hole in the middle
    drawDot(CCPointZero, ENEMY_RADIUS, ccc4f(0, 0, 0, 1));

    setScale(0.0f);

    return true;
}
```

We begin by calling the init function of the parent class and storing a reference to GameWorld. We then initialize an array of the type CCPtint and pass it into the GenerateVertices function. Here, NUM_SPIKES equals 10. We then call the drawPolygon function followed by the drawDot function to create a red colour-filled polygon and a black colour-filled circle, giving the enemies their deadly appearance.

Now let's take a look at the GenerateVertices function:

```cpp
void Enemy::GenerateVertices(CCPoint vertices[])
{
    Generate Vertices (vertices[]);
}
```
vector<CCPoint> inner_vertices, outer_vertices;
// get two regular polygons, one smaller than the other and with a
slightly advance rotation
GameGlobals::GetRegularPolygonVertices(inner_vertices,
NUM_SPIKES, ENEMY_RADIUS);
GameGlobals::GetRegularPolygonVertices(outer_vertices, NUM_SPIKES,
ENEMY_RADIUS * 1.5f, M_PI / NUM_SPIKES);

// run a loop to splice the polygons together to form a star
for(int i = 0; i < NUM_SPIKES; ++i)
{
    vertices[i*2] = inner_vertices[i];
    vertices[i*2 + 1] = outer_vertices[i];
}

This function simply creates two vectors for the inner and outer vertices that make
up the enemy's spiky appearance. Furthermore, you can see that we fill up these two
vectors with vertices generated by the GetRegularPolygonVertices function that
we defined earlier. Notice how the radius for the outer vertices is 1.5 times the radius
for the inner vertices. We then run a loop and fill up the input array of CCPoint with
inner and outer vertices, respectively. This order of inner vertex followed by outer
vertex is important because otherwise we wouldn't get a properly filled polygon.
You should reverse the order and see for yourself what happens to gain a better
understanding of how the colour is filled inside a convex polygon.

Let's now give the enemies their behavior by defining the Update function:

void Enemy::Update(CCPoint player_position, bool towards_player)
{
    // no movement while spawning
    if(is_spawning_)
        return;

    // first find a vector pointing to the player
    CCPoint direction = ccpSub(player_position, m_obPosition);
    // normalize direction then multiply with the speed_multiplier_
    speed_ = ccpMult(direction.normalize(), speed_multiplier_ * (towards_player ? 1 : -1));

    // restrict movement within the boundary of the game
    CCPoint next_position = ccpAdd(m_obPosition, speed_);
    if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_, next_position,
        ENEMY_RADIUS * 1.5f))
    
    
    
}
setPosition(next_position);
}
else
{
    if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_, CCPoint(
        next_position.x - speed_.x, next_position.y), ENEMY_RADIUS * 
        1.5f))
    {
        setPosition(ccp(next_position.x - speed_.x, next_position.y));
    }
    else if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_, CCPoint( 
        next_position.x, next_position.y - speed_.y), ENEMY_RADIUS * 
        1.5f))
    {
        setPosition(ccp(next_position.x, next_position.y - speed_.y));
    }
}

The Update function, called from the main update loop by GameWorld, is passed the player's position and whether the movement should be toward or away from the player. Why would such a deadly, fearless enemy be moving away from the player you wonder? Well, that happens when the player has the shield power-up enabled of course! Each enemy also has a spawning animation defined in the Spawn function, during which it is not supposed to move—hence the conditional and the return statements. We first calculate the speed for the enemy based on the direction towards or away from the player and a speed multiplier. Next, we perform some bounds checking and ensure that the enemy does not leave the boundary that is defined by boundary_rect_ inside GameWorld. To perform this boundary check, we define a resourceful function in GameGlobals.h with the name RECT_CONTAINS_CIRCLE.

I'm sure you remember how the enemies are supposed to move faster the longer they live. This behavior is defined in the Tick function of the Enemy class:

```cpp
void Enemy::Tick()
{
    // no ticking while spawning
    if(is_spawning_)
        return;

    ++ time_alive_;

    // as time increases, so does speed
    switch(time_alive_)
    {
```
case E_SLOW:
    speed_multiplier_ = 0.5f;
    break;
case E_MEDIUM:
    speed_multiplier_ = 0.75f;
    break;
case E_FAST:
    speed_multiplier_ = 1.25f;
    break;
case E_SUPER_FAST:
    speed_multiplier_ = 1.5f;
    break;
}

The speed multiplier that you saw inside the Enemy::Update function is given its value in the Tick function. We update the variable time_alive_ that measures the seconds for which the enemy has been alive and, based on its value, assign a particular speed value to the speed_multiplier_ variable so the enemy moves faster the longer that it lives. To enumerate the various speeds an enemy may move at, an enum by the name of EEnemySpeedTimer is defined in GameGlobals.h as follows:

enum EEnemySpeedTimer
{
    E_SLOW = 5,
    E_MEDIUM = 10,
    E_FAST = 15,
    E_SUPER_FAST = 25,
};

We now have a player and an enemy that follows the player at increasing speeds. But the player still has no way to defend himself or to defeat the enemies. So let's define the last set of elements, the power-ups.
The PowerUp class

The power-ups in this game constitute a bomb, a missile launcher, and a shield. All three of these power-ups have some behavior in common. They all have icons with which the player must collide to activate them, they all stay on screen for a specific time after which they must disappear, and they can all be activated or deactivated. We separate these behaviors in a parent class named `PowerUp` as follows:

```cpp
#ifndef POWERUP_H_
#define POWERUP_H_

#include "GameGlobals.h"

class GameWorld;

class PowerUp : public CCDrawNode
{
public:
    PowerUp() : time_left_(0), speed_(CCPointZero), is_active_(false), must_be_removed_(false), game_world_(NULL){}
    ~PowerUp(){};

    virtual bool init(GameWorld* instance);  
    virtual void Update();  
    virtual void Tick();  
    virtual void Spawn();  
    virtual void Activate();  
    virtual void Deactivate();

    CC_SYNTHESIZE(int, time_left_, TimeLeft);
    CC_SYNTHESIZE(CCPoint, speed_, Speed);
    CC_SYNTHESIZE(bool, is_active_, IsActive);
    CC_SYNTHESIZE(bool, must_be_removed_, MustBeRemoved);

protected:
    GameWorld* game_world_; 
};

#endif // POWERUP_H_
```
The PowerUp class will extend the CCDrawNode class, since all the power-ups will need primitive drawing functionality. This class also defines a few variables to keep track of the time on screen, the movement speed, whether the power-up is active, and finally whether it should be removed by GameWorld. The main lifecycle functions are declared here and marked as virtual so the child classes can override them to implement their own respective behaviors. Let's now take a look at some of these functions defined in PowerUp.cpp:

```cpp
bool PowerUp::init(GameWorld* instance)
{
    if(!CCDrawNode::init())
        return false;

    game_world_ = instance;
    // calculate how much time the power-up should wait on screen before activation
    time_left_ = MAX_POWERUP_WAIT_ON_SCREEN / 2 + CCRANDOM_0_1() * MAX_POWERUP_WAIT_ON_SCREEN / 2;
    // calculate speed
    speed_ = CCPoint(CCRANDOM_MINUS1_1() * 2,
                     CCRANDOM_MINUS1_1() * 2);

    // draw the brown coloured ring
drawDot(CCPointZero, POWERUP_ICON_OUTER_RADIUS,
           ccc4f(0.73725f, 0.5451f, 0, 1));
drawDot(CCPointZero, POWERUP_ICON_OUTER_RADIUS - 3,
           ccc4f(0, 0, 0, 1));
    setScale(0.0f);

    return true;
}
```

The init function basically takes care of maintaining a reference to GameWorld for ease of access and initializing a few variables. Each power-up comes to life with its respective icon, but all those icons have their outer ring in common with each other, so we define that here in the parent class. We also set the scale initially to 0 since we will be animating the birth of each power-up in the Spawn function. We now look at the Update function that will be called at every tick by GameWorld:

```cpp
void PowerUp::Update()
{
    // bounce within the boundary
    if(!RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_, m_obPosition,
                               POWERUP_ICON_OUTER_RADIUS))
    {
```
// bounce off the left & right edge
if( (m_obPosition.x - POWERUP_ICON_OUTER_RADIUS) <
    game_world_->boundary_rect_.origin.x ||
    (m_obPosition.x + POWERUP_ICON_OUTER_RADIUS) > (game_world_->boundary_rect_.origin.x +
    game_world_->boundary_rect_.size.width) )
speed_.x *= -1;

// bounce off the top & bottom edge
if( (m_obPosition.y + POWERUP_ICON_OUTER_RADIUS) > (game_world_->boundary_rect_.origin.y +
    game_world_->boundary_rect_.size.height) ||
    (m_obPosition.y - POWERUP_ICON_OUTER_RADIUS) <
    game_world_->boundary_rect_.origin.y )
speed_.y *= -1;
}

setPosition(m_obPosition.x + speed_.x, m_obPosition.y + speed_.y);
}

This function basically takes care of moving the power-up within the boundary defined in GameWorld (similar to the Enemy class) by using the RECT_CONTAINS_CIRCLE function.

Let's take a look at the following code:

void PowerUp::Tick()
{
    -- time_left_;

    // remove this power-up in the next iteration when it's on-screen
time is over
    if(time_left_ < 0)
    {
        must_be_removed_ = true;
        runAction(CCSequence::createWithTwoActions(CCEaseBackIn::create(
            CCScaleTo::create(0.25f, 0.0f)), CCRemoveSelf::create(true)));
    }
}

Each power-up must also have a life time after which it should be removed. This happens in the Tick function that is called by GameWorld once every second.
Let's take a look at the following code:

```cpp
void PowerUp::Activate()
{
    // clear the geometry and stop all actions
    // now the child classes can add their own behaviour
    is_active_ = true;
    clear();
    stopAllActions();
}

void PowerUp::Deactivate()
{
    // remove this power-up in the next iteration
    runAction(CCSequence::createWithTwoActions(
        CCDelayTime::create(0.01f), CCRemoveSelf::create(true)));
    must_be_removed_ = true;
}
```

Finally, we have the `Activate` and `Deactivate` functions that set the appropriate flags and prepare the power-up for whatever behaviour the child class may define. Notice how the `clear` function is called in the `Activate` method. This happens because the power-up is initially nothing but an icon and must turn into its respective manifestation now that it has been triggered or activated. Hence, we call the `clear` function of parent class `CCDrawNode`, which basically clears all the geometry drawn inside the node so far. Now that we have the parent class defined, we can take the time to define each power-up separately starting with the `Bomb` class, followed by the `MissileLauncher` class and `Shield` class.

**The Bomb class**

The behavior of the `Bomb` class is quite simple: when triggered, this power-up creates a big explosion that stays on screen for a couple of seconds. All enemies that come in contact with this explosion die a miserable, fiery death. Let's begin by defining the `init` function of this power-up:

```cpp
bool Bomb::init(GameWorld* instance)
{
    if(!PowerUp::init(instance))
        return false;
```
// get vertices for a triangle
vector<CCPoint> vertices;
GameGlobals::GetRegularPolygonVertices(vertices, 3,
    POWERUP_ICON_INNER_RADIUS);
// draw a triangle with a green border
drawPolygon(&vertices[0], 3, ccc4f(0, 0, 0, 0), 3, ccc4f(0, 1, 0, 1));

    return true;
}

Right at the beginning, we must invoke the init function of the parent class and pass in the reference to GameWorld. Once that is done, we can generate the individual icon for this power-up, which in this case is nothing but a green coloured triangle. Let's wrap up the Bomb by overriding the Activate function:

    void Bomb::Activate()
    {
        // must activate only once
        if(is_active_)
            return;

        // first call parent function
        PowerUp::Activate();

        // create a blast 8 times the size of the player that should last for 2 seconds
        Blast* blast = Blast::createWithRadiusAndDuration(
            PLAYER_RADIUS * 8, 2.0f);
        // position blast over bomb
        blast->setPosition(m_obPosition);
        game_world_->AddBlast(blast);
        SOUND_ENGINE->playEffect("big_blast.wav");

        PowerUp::Deactivate();
    }
This function must only be called once, hence the return statement at the beginning. We then immediately call the `Activate` function of the parent class, since we want the geometry cleared. Now comes the interesting part. This is where we actually create the explosion. This is taken care of by another class: `Blast`. Why couldn't we just create an explosion within this perfectly capable `Bomb` class itself? The reason will become abundantly clear when we define the behaviour of the missiles. For now, all I can tell you is that just like bombs, missiles also explode and hence that functionality is shared by the two and separated out. We then set the position of this blast over the bomb and finally hand it over to `GameWorld`. We also play a sound effect before deactivating the bomb. Let's now glance at the behaviour of the `Blast` class.

**The Blast class**

A blast represents nothing but an explosion that obliterates all enemies that it comes in contact with. The `Blast` class will inherit from `CCDrawNode` and will use a set of circles to draw an explosion in the `initWithRadiusAndDuration` function:

```cpp
bool Blast::initWithRadiusAndDuration(float radius, float duration) {
    if (!CCDrawNode::init()) {
        return false;
    }

    radius_ = radius;
    duration_ = duration;

    // initially scale down completely
    setScale(0.0f);
    drawDot(CCPointZero, radius_, ccc4f(1, 0.34118f, 0, 1));
    drawDot(CCPointZero, radius_ * 0.8f, ccc4f(1, 0.68235f, 0, 0.25f));
    drawDot(CCPointZero, radius_ * 0.75f, ccc4f(1, 0.68235f, 0, 0.5f));
    drawDot(CCPointZero, radius_ * 0.7f, ccc4f(1, 0.68235f, 0, 0.5f));
    drawDot(CCPointZero, radius_ * 0.6f, ccc4f(1, 0.83529f, 0.40392f, 0.25f));
    drawDot(CCPointZero, radius_ * 0.55f, ccc4f(1, 0.83529f, 0.40392f, 0.5f));
    drawDot(CCPointZero, radius_ * 0.5f, ccc4f(1, 0.83529f, 0.40392f, 0.5));
    drawDot(CCPointZero, radius_ * 0.4f, ccc4f(1, 1, 1, 0.25f));
    drawDot(CCPointZero, radius_ * 0.35f, ccc4f(1, 1, 1, 0.75f));
    drawDot(CCPointZero, radius_ * 0.3f, ccc4f(1, 1, 1, 1));
```
// scale-up, then wait for 'duration_' amount of seconds before cooling down
runAction(CCSequence::create(CC.easeSineOut::create(
    CCScaleTo::create(0.25f, 1.0f)), CCDelayTime::create(duration_),
    CCCallFunc::create(this, callfunc_selector(Blast::Cooldown)),
    NULL));

return true;
}

The initWithRadiusAndDuration function saves the blast radius that GameWorld will use for collision detection and also the duration after which this blast should cool down. We then draw circles of different colors to represent our vibrant explosion. We animate its entry and use the CCDelayTime to keep the explosion active for the time specified by duration_ before finally telling it to cool down.

Let's take a look at the following code:

void Blast::Cooldown()
{
    // remove this blast in the next iteration
    must_be_removed_ = true;
    // animate exit then remove with cleanup
    runAction(CCSequence::createWithTwoActions(CC.easeSineOut::create(
        CCScaleTo::create(0.5f, 0.0f)), CCRemoval::create(true)));
}

The cool down involves an exit animation and enabling the must_be_removed_flag so that GameWorld will remove this Blast in the next iteration.

The MissileLauncher class
The behavior of the MissileLauncher class is to spawn five missiles upon activation by the player. This class should also assign a target for each missile to hit and hand them over to GameWorld. This is how its init function looks inside MissileLauncher.cpp:

bool MissileLauncher::init(GameWorld* instance)
{
    if(!PowerUp::init(instance))
        return false;

    vector<CCPoint> vertices1;
    vector<CCPoint> vertices2;
vector<CCPoint> vertices;

// get two regular pentagons, one smaller than the other and with a slightly advance rotation
GameGlobals::GetRegularPolygonVertices(vertices1, 5,
    POWERUP_ICON_INNER_RADIUS - 6, M_PI * -2/20);
GameGlobals::GetRegularPolygonVertices(vertices2, 5,
    POWERUP_ICON_INNER_RADIUS, M_PI * 2/20);

// run a loop to splice the pentagons together to form a star
for(int i = 0; i < 5; ++i)
{
    vertices.push_back(vertices1[i]);
    vertices.push_back(vertices2[i]);
}

// draw the star shaped polygon with yellow border
drawPolygon(&vertices[0], 10, ccc4f(0, 0, 0, 0), 2,
    ccc4f(0.88235, 0.96078, 0, 1));

return true;
}

The init method for all power-ups takes care of creating the icon, so we begin by generating the vertices for this particular power-up’s icon. Notice how we fetch vertices for two regular pentagons here and then splice them together into another vector. We then pass that vector to the drawPolygon function to get the yellow star with a border of 2 pixels. If you’re confused, the following images will clear things up:

<table>
<thead>
<tr>
<th>Inner Pentagon (vertices1)</th>
<th>Outer Pentagon (vertices2)</th>
<th>Inner and Outer Pentagon (vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Inner Pentagon" /></td>
<td><img src="image2" alt="Outer Pentagon" /></td>
<td><img src="image3" alt="Inner and Outer Pentagon" /></td>
</tr>
</tbody>
</table>
Now, let's define what happens when the missile launcher has been activated in the `Activate` function:

```cpp
void MissileLauncher::Activate()
{
    if(is_active_)
        return;

    PowerUp::Activate();

    // generate a target for each missile
    vector<CCPoint> target = GenerateTargets();
    // generate an initial direction vector for each missile
    vector<CCPoint> initial_direction;
    GameGlobals::GetRegularPolygonVertices(initial_direction, 5,
                                            SCREEN_SIZE.width/4, M_PI * 2/20);

    for(int i = 0; i < 5; ++i)
    {
        // create a missile with a target, initial direction & speed
        Missile* missile = Missile::createWithTarget(game_world_,
                                                   target[i],
                                                   ccpMult(initial_direction[i].normalize(), MISSILE_SPEED));
        // position the missile over the launcher
        missile->setPosition(m_obPosition);
        game_world_->AddMissile(missile);
    }

    SOUND_ENGINE->playEffect("missile.wav");

    PowerUp::Deactivate();
}
```

The `MissileLauncher` class is supposed to spawn five missiles that fly towards enemies and explode on contact. Thus, the first thing we need to do is generate targets and initial directions for each missile. We then run a loop to create five `Missile` objects and pass in a reference to `GameWorld`, the target position, and the initial speed. The initial speed is a scalar multiplication of `initial_direction` with a constant called `MISSILE_SPEED`. 
We set the position for each missile and hand them over to GameWorld. We wind up this missile launcher by playing a sound effect and finally deactivate this power-up since its job is done. Before we move to the Missile class, let’s take a look at the GenerateTargets function:

```cpp
vector<CCPoint> MissileLauncher::GenerateTargets()
{
    vector<CCPoint> target_points;
    target_points.clear();
    int targets_found = 0;

    int num_enemies = game_world_->enemies_->count();
    // loop through the first 5 enemies within GameWorld &
    save their positions
    for(int i = 0; i < num_enemies && targets_found < 5; ++i)
    {
        Enemy* enemy = (Enemy*)game_world_->enemies_->objectAtIndex(i);
        target_points.push_back(enemy->getPosition());
        ++targets_found;
    }
    // if less than 5 enemies were found, fill up with random positions
    // within the boundary
    while(targets_found < 5)
    {
        target_points.push_back(CCPoint(CCRANDOM_0_1() * (game_world_->boundary_rect_.origin.x + game_world_->boundary_rect_.size.width), CCRANDOM_0_1() * (game_world_->boundary_rect_.origin.y + game_world_->boundary_rect_.size.height)));
        ++targets_found;
    }
    return target_points;
}
```

This function returns a vector of CCPoint, which will represent the target for a given missile. So, we loop over the enemies currently held by GameWorld and save the position of the first five enemies we find. However, if we don’t have enough targets after this loop, the missile still must fire to somewhere. Thus, we send that missile to any random point within the boundary of the game. So, we have launched five missiles with a target to hunt down and kill. Let’s now take a look at the behaviour of a missile by defining the Missile class.
The Missile class

The behaviour of the missiles is to fly towards their assigned target points and explode. However, we want it to look cool so we define the behaviour of these missiles such that they follow a smooth curved path towards their targets. Now, let's define their creation in the initWithTarget function:

```cpp
bool Missile::initWithTarget(GameWorld* instance, CCPoint target, CCPoint speed)
{
    if(!CCDrawNode::init())
    {
        return false;
    }

    game_world_ = instance;
    target_ = target;
    speed_ = speed;

    // generate vertices for the missile
    CCPoint vertices[] = {
        CCPoint(MISSILE_RADIUS * 1.75f, 0),
        CCPoint(MISSILE_RADIUS * -0.875f, MISSILE_RADIUS),
        CCPoint(MISSILE_RADIUS * -1.75f, 0),
        CCPoint(MISSILE_RADIUS * -0.875f, MISSILE_RADIUS * -1)};

    // draw a yellow coloured missile
    drawPolygon(vertices, 4, ccc4f(0.91765f, 1, 0.14118f, 1), 0,
                 ccc4f(0, 0, 0, 0));

    // schedule to explode after 5 seconds
    scheduleOnce(schedule_selector(Missile::Explode), 5.0f);
    scheduleUpdate();

    return true;
}
```

The Missile class will also inherit CCDrawNode, which means we must call the parent class' init function. We save a reference to GameWorld, the target point, and the initial speed that this missile should move at. We then define the missile's visual representation by creating an array of vertices and passing them to the drawPolygon function. This results in a yellow color-filled missile that is similar in shape to the Player entity. Finally, we schedule the Explode function to be called after 5.0f seconds. This is because we want this missile to automatically explode after sometime if it hasn't collided with any enemies.
We wind up this function by scheduling the `update` function, which we will define as follows:

```cpp
void Missile::update(float dt)
{
    // find a vector pointing to the target
    CCPoint direction = ccpSub(target_, m_obPosition).normalize();
    // add the direction to the speed for smooth curved movement
    speed_.x += direction.x;
    speed_.y += direction.y;
    // normalize the speed & multiply with a constant
    speed_ = ccpMult(speed_.normalize(), MISSILE_SPEED);
    setPosition(m_obPosition.x + speed_.x, m_obPosition.y + speed_.y);
    // update the rotation of the missile
    float angle = ccpToAngle(ccpSub(m_obPosition, previous_position_));
    setRotation(CC_RADIANS_TO_DEGREES(angle * -1));
    previous_position_ = m_obPosition;
    // explode the missile if it has roughly reached the target
    if(m_obPosition.fuzzyEquals(target_, ENEMY_RADIUS * 1.5f))
    {
        Explode();
    }
}
```

The `update` function is responsible for updating the position and rotation of the missile. Thus, we calculate the difference between the target and the missile's current position. Normalizing this `CCPoint` will give us the direction in which this missile must move in order to hit its target. We then add this direction to `speed_` because we want the missile to move in a smooth curve towards its target. Finally, we normalize `speed_` and multiply it with a constant, `MISSILE_SPEED`. If we don't do this last step, the missile's speed will keep increasing and it will never reach its target.
We also set the rotation of this missile using the useful ccpToAngle function, passing in the current and previous positions. To wrap up the update function, we check whether the missile has reached its target by using the fuzzyEquals function of the CCPoint class. This function basically adds some tolerance to the equality checking of two points. In this case, this tolerance is the outer radius of an enemy. If a collision is detected, we order this missile to explode in the Explode function, which we will define as follows:

```cpp
void Missile::Explode(float dt)
{
    // can't explode more than once
    if(has_exploded_)
        return;

    has_exploded_ = true;
    // create three blasts on explosion
    for(int i = 0; i < 3; ++i)
    {
        // create a blast twice the size of the player that should last
        // quarter of a second
        Blast* blast = Blast::createWithRadiusAndDuration(
            PLAYER_RADIUS * 2, 0.25f);
        // position it randomly around the missile
        blast->setPosition(ccpAdd(m_obPosition,
            CCPoint(CCRANDOM_0_1() * PLAYER_RADIUS * 2 * i,
                CCRANDOM_0_1() * PLAYER_RADIUS * 2 * i)));
        game_world_->AddBlast(blast);
    }
    // remove this missile in the next iteration
    must_be_removed_ = true;
    runAction(CCSequence::createWithTwoActions(
        CCDelayTime::create(0.01f), CCRemoveSelf::create(true)));
    SOUND_ENGINE->playEffect("small_blast.wav");
}
```

A missile might explode on three different occasions; when it has run out of time, when it has reached its target, or when it has collided with an enemy on the way to its target. We must ensure that a missile must explode only once and hence we enable the has_exploded_flag to prevent further explosions. Each missile explodes and results in three blasts. In a loop, we create three Blast objects, position them, and hand them over to GameWorld. Notice that these blasts are smaller and stay for much lesser time than the blast created by the Bomb. We finally finish the Explode function by enabling the must_be_removed_flag, running a CCRemoveSelf action, and playing a sound effect. That wraps up our Missile class and it is time to define our last power-up, and my personal favorite, the Shield class.
The Shield class

The shield is without a doubt the most powerful of all the power-ups. That's because this shield not only protects the player from contact with the enemies, but also kills all enemies that come in contact with it. Now, the enemies know this and start moving away from the player the moment the shield is activated. Don't tell me you still like the MissileLauncher the most. The shield lasts for a whole of 10 seconds before it disables. The Shield class will override the Tick function in addition to the Update, Activate, and Deactivate functions from the PowerUp parent class.

The init function for the Shield class simply creates a cyan hexagon as the icon, and hence we will skip straight to the Update function that is called at every tick from GameWorld:

```cpp
void Shield::Update()
{
    if(!is_active_)
    {
        PowerUp::Update();
    }
    else
    {
        // after activation, shield will follow the player
        setPosition(game_world_->player_->getPosition());
        setRotation(game_world_->player_->getRotation());
    }
}
```

The behavior of this power-up is different before and after activation. Before activation, the shield is like every other power-up: it just floats around the screen. After activation, however, the shield must stick to the player and rotate along with the player. Next up is the Tick function that is called by GameWorld once every second:

```cpp
void Shield::Tick()
{
    if(is_active_)
    {
        -- shield_time_left_;  
        // deactivate the shield when it's time is over
        if(shield_time_left_ <= 0)
        {
```
Before activation, the shield is dormant just like the other power-ups so we call the `Tick` function of the parent class. But after activation, we monitor how much time the shield has left till it must be disabled by the updating variable `shield_time_left_`. Initially, `shield_time_left_` is set to 10 and is decremented every `Tick`. When the value hits 0, we `Deactivate` the shield. When the value hits 2, we start blinking the shield so the user knows that shield is about to be disabled and that there are just 2 seconds of carnage left. We set a tag for the blinking action that will come in handy later.

Let's take a look at the following code:

```cpp
void Shield::Activate()
{
    if(is_active_)
        return;

    // if a shield already exists on the player,
    if(game_world_->player_->GetShield())
    {
        // reset the existing shield
        game_world_->player_->GetShield()->Reset();
        // deactivate self
        Deactivate();
        removeFromParentAndCleanup(true);
    }
}
```
We first check if the player has a shield already enabled. Why should we do that? This is because we can't have two shields active at the same time—that would look silly! Instead, we reset the first shield and discard the second one.

Thus, if a shield is already active on the player, we disable this shield by calling the Deactivate method and call removeFromParentAndCleanup. Also, we call the Reset function of the shield currently active on the player, which looks like this:

```cpp
void Shield::Reset()
{
    // reset the shield duration
    shield_time_left_ = SHIELD_DURATION;
    // stop any blinking action & show the shield if it was hidden due to the blink
    stopActionByTag(SHIELD_BLINK_TAG);
    setVisible(true);
}
```

As you can see, we simply reset the duration of the shield and stop any blinking action. That last setVisible is there to undo any invisibility due to the blinking.

Now, let's move back to the Activate function:

```cpp
// else if shield doesn't exist on the player
else
{
    PowerUp::Activate();

    // set the shield duration
    shield_time_left_ = SHIELD_DURATION;
    setScale(0);

    // generate & draw a bigger cyan hexagon
    vector<CCPoint> vertices;
    GameGlobals::GetRegularPolygonVertices(vertices, 6,
        PLAYER_RADIUS * 2.5f);
    drawPolygon(&vertices[0], 6, ccc4f(0, 0, 0, 0), 4,
        ccc4f(0, 0.96862f, 1, 1));

    // animate the activation & life of the shield
    runAction(CCEaseBounceOut::create(CCScaleTo::create(0.25f, 1.0f)));
    runAction(CCRepeatForever::create(CCSequence::createWithTwoActions(
        CCEaseSineOut::create(CCScaleTo::create(0.25f, 1.15f)),
        CCEaseSineOut::create(CCScaleTo::create(0.25f, 1.0f)))));
```
// inform the player that it now has a shield around it
game_world_->player_->SetShield(this);

SOUND_ENGINE->playEffect("shield.wav");
}

If there is no shield active on the player, we call the Activate function of the parent class and initialize the shield's duration again. We also set the scale to 0, since we want to animate the activation of the shield. Since PowerUp::Activate clears the node's geometry, we generate vertices for a hexagon that's big enough to cover the player. We then repeat a simple scale-up or scale-down animation and finally inform the player that it has a shield around it. Outside the if-else block, we play a sound effect marking the activation of the shield and finish off the Activate function.

In the Deactivate function (not shown in the code), we simply inform the player that it doesn't have the shield around it any more. With that, we wrap up our power-ups and can now move on to the game world.

Creating the game

Let's define the CreateGame function, which is called from the init function when GameWorld is created:

```cpp
void GameWorld::CreateGame()
{
    // initialise counters & flags
    seconds_ = 0;
    enemies_killed_total_ = 0;
    enemies_killed_combo_ = 0;
    combo_timer_ = 0;
    score_ = 0;
    is_popup_active_ = false;

    // add the stars
    background_ = BackgroundManager::create();
    addChild(background_, E_LAYER_BACKGROUND);

    CreateBoundary();
    CreatePlayer();
    CreateContainers();
    CreateHUD();
```
// initially add some enemies & a powerup
AddEnemyFormation();
AddPowerUp();

// schedule the update and the tick
scheduleUpdate();
schedule(schedule_selector(GameWorld::Tick), 1.0f);
}

We start this function by initializing the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds_</td>
<td>This is the amount of time the game has been active.</td>
</tr>
<tr>
<td>enemies_killed_total_</td>
<td>This is the total number of enemies killed.</td>
</tr>
<tr>
<td>enemies_killed_combo_</td>
<td>This is the number of enemies killed in the last 3 seconds.</td>
</tr>
<tr>
<td>combo_timer_</td>
<td>This counts down from 3 seconds every time an enemy is killed.</td>
</tr>
<tr>
<td>score_</td>
<td>This is the score earned by killing enemies and from combos.</td>
</tr>
<tr>
<td>is_popup_active_</td>
<td>This is the flag that is used to pause/start when a popup is activated/deactivated.</td>
</tr>
</tbody>
</table>

The function then proceeds to create and add the BackgroundManager. We will not discuss the BackgroundManager class, but it basically creates a CCDrawNode that draws a number of white circles with varying opacities to serve as a starry space background for the game.

The function then calls CreateBoundary, which creates a CCDrawNode with a semi-transparent white rectangle drawn inside it to represent the bounds of the play area. The CreatePlayer functions simply creates an object of the Player class and positions it in the center of the play area before adding it to GameWorld. The CreateContainers function creates and retains four arrays: enemies_, powerups_, blasts_, and missiles_ to hold the Enemy, PowerUp, Blast, and Missile objects, respectively. The CreateHUD function simply creates and adds a CLabelBMFont to display the score.

We also add a formation of enemies along with a power-up to get the user started. Finally, we schedule the update function and the Tick function to be called once every second.
The update loop

The update function for Inverse Universe will be similar to most other games where we update all the game elements and check for collisions. The code looks like this:

```cpp
void GameWorld::update(float dt)
{
    // don't process if player is dying
    if(player_->is_dying_)
        return;

    // update each enemy
    CCObject* object = NULL;
    CCARRAY_FOREACH(enemies_, object)
    {
        Enemy* enemy = (Enemy*)object;
        if(enemy)
        {
            enemy->Update(player_->getPosition(), player_->GetShield() == NULL);
        }
    }

    // update each power-up
    object = NULL;
    CCARRAY_FOREACH(powerups_, object)
    {
        PowerUp* powerup = (PowerUp*)object;
        if(powerup)
        {
            powerup->Update();
        }
    }

    CheckCollisions();
    CheckRemovals();
}
```

We skip processing anything if the player's death animation is playing. We then iterate over the enemies_ and powerups_ arrays and update each object they contain. Finally, we check for collisions and for objects that must be removed. We will skip discussing the CheckRemovals function but to give you a gist, it basically checks the state of the must_be_removed_ flag for Enemy, PowerUp, Blast, and Missile and removes them if the flag is enabled.
Let's now look at the `CheckCollisions` function. This function will basically use circular collision detection by using the `CIRCLE_INTERSECTS_CIRCLE` function defined in `GameGlobals.h`. All you need to do is provide the center and radius of the two circles and the function returns `true` if a collision is found.

Let's take a look at the following code:

```cpp
void GameWorld::CheckCollisions()
{
    // save player position & radius
    CCPoint player_position = player_->getPosition();
    float player_radius = player_->getRadius();

    // iterate through all enemies
    CCObject* object = NULL;
    CCARRAY_FOREACH(enemies_, object)
    {
        Enemy* enemy = (Enemy*)object;
        if(enemy)
        {
            CCPoint enemy_position = enemy->getPosition();

            // check with Player
            if(CIRCLE_INTERSECTS_CIRCLE(player_position, player_radius,
                                         enemy_position, ENEMY_RADIUS))
            {
                // if shield is enabled, kill enemy
                if(player_->GetShield())
                {
                    enemy->Die();
                    EnemyKilled();
                }

                // else kill player...but only if enemy has finished spawning
                else if(!enemy->getIsSpawning())
                {
                    player_->Die();
                }
            }
        }
    }
}
```

A lot of the collision detection revolves around the enemies. So, we first check for collisions of each enemy with the player. If a collision is found, we either kill the player or the enemy based on the shield being enabled or disabled.

Another thing to consider is how we check whether the enemy is spawning and skip killing the player. We do this on purpose, or else the player will die before the enemy's spawning is complete—thereby leaving the user bewildered about the player's death.
Let's take a look at the following code:

```c
// check with all blasts
CCObject* object2 = NULL;
CCARRAY_FOREACH(blasts_, object2)
{
    Blast* blast = (Blast*)object2;
    if(blast)
    {
        if(CIRCLE_INTERSECTS_CIRCLE(blast->getPosition(),
                                      blast->getRadius(),
                                      enemy_position, ENEMY_RADIUS*1.5f))
        {
            enemy->Die();
            EnemyKilled();
        }
    }
}

// check with all missiles
object2 = NULL;
CCARRAY_FOREACH(missiles_, object2)
{
    Missile* missile = ( Missile* ) object2;
    if( missile )
    {
        if( CIRCLE_INTERSECTS_CIRCLE( missile->getPosition(),
                                        MISSILE_RADIUS, enemy_position, ENEMY_RADIUS*1.5f) )
        {
            missile->Explode();
        }
    }
}
```

We then proceed to check collisions of the enemy with the blasts and kill any enemy coming in contact with the blast. We also explode any missile that has come in contact with any enemy.
Let's take a look at the following code:

```cpp
// check if player collides with any of the power-ups  
// activate the power-up if collision is found
object = NULL;
CCARRAY_FOREACH(powerups_, object)
{
    PowerUp* powerup = (PowerUp*)object;
    if(powerup && !powerup->getIsActive())
    {
        if(CIRCLE_INTERSECTS_CIRCLE(player_position, player_radius,
                                    powerup->getPosition(), POWERUP_ICON_OUTER_RADIUS))
        {
            powerup->Activate();
        }
    }
}

Finally, we check for collisions between the player and the power-ups and activate the power-up on contact with the player. That wraps up the CheckCollisions function but before we move to the Tick function, let's take a look at the EnemyKilled function:

```cpp
void GameWorld::EnemyKilled()
{
    // increment counters
    ++ enemies_killed_total_;  
    ++ enemies_killed_combo_;  
    // reset combo time
    combo_timer_ = COMBO_TIME;

    // add score & update the label
    score_ += 7;
    char buf[16] = {0};
    sprintf(buf, "Score: %d", score_);  
    score_label_->setString(buf);
}
Quite simply, we increment the total number of enemies killed as well as the number of enemies killed in the current combo. We can then use the number of enemies killed in the current combo to reward the player with some extra points when the combo timer elapses. We also reset the combo timer, increment the score, and update the HUD. Next up is the Tick function:

```cpp
void GameWorld::Tick(float dt)
{
    // don't tick if player is dying
    if(player_->is_dying_) return;

    ++ seconds_;
    -- combo_timer_;  // show the combo achieved if time is up
    if(combo_timer_ < 0)
        combo_timer_ = 0;
    else if(combo_timer_ == 0)
        ComboTimeUp();

    // Tick each enemy
    CCObject* object = NULL;
    CCARRAY_FOREACH(enemies_, object)
    {
        Enemy* enemy = (Enemy*)object;
        if(enemy)
            enemy->Tick();
    }

    // Tick each power-up
    object = NULL;
    CCARRAY_FOREACH(powerups_, object)
    {
        PowerUp* powerup = (PowerUp*)object;
        if(powerup)
            powerup->Tick();
    }
}
```
// add an enemy formation every 5 seconds
if(seconds_ % 5 == 0)
    AddEnemyFormation();
// add a powerup formation every 4 seconds
if(seconds_ % 4 == 0)
    AddPowerUp();
}

The first thing to do is to increment the seconds_counter. The use of this variable will become clear in the last section when we use it to make the game progressively more difficult. Next, we decrement the combo_timer_ and call the ComboTimeUp function that displays the number of enemies killed by the player in the last 3 seconds.

Then, we call the Tick function for the Enemy and PowerUp objects. Finally, we add a new formation of enemies every 5 seconds and a new power-up every 4 seconds.

Adding tilt controls

Tilt controls, also referred to as the accelerometer, can be added to any layer in the same way touch controls are added. It is as simple as calling the setAccelerometerEnabled(true) function in the init function of GameWorld and then overriding the virtual function didAccelerate. Let's take a look at the didAccelerate function:

```cpp
void GameWorld::didAccelerate(CCAcceleration* acceleration_value)
{
    HandleInput(ccp(acceleration_value->x, acceleration_value->y));
}
```

The didAccelerate function gets a parameter of type CCAcceleration*, which contains three values: x, y, and z. These three values signify how much the device has tilted in the x, y, and z directions. Take a look at the following table to understand the values Cocos2d-x passes into the didAccelerate function:

<table>
<thead>
<tr>
<th>Tilt direction</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extreme left</td>
<td>-0.9f to -1.1f</td>
<td>0.0f</td>
</tr>
<tr>
<td>Extreme right</td>
<td>0.9f to 1.1f</td>
<td>0.0f</td>
</tr>
<tr>
<td>Extreme forward</td>
<td>0.0f</td>
<td>0.9f to 1.1f</td>
</tr>
<tr>
<td>Extreme backward</td>
<td>0.0f</td>
<td>-0.9f to -1.1f</td>
</tr>
</tbody>
</table>
The preceding values are approximate and will differ a bit from device to device.

For the purpose of our game, we will need just horizontal and vertical movement, so we only use the x and y members of `acceleration_value`.

I'm sure you were expecting more, but the reason for separating the logic into the `HandleInput` function is because this game was first built on `win32` and the tilt controls were tested on an Android device later. So, I needed to put in touch controls for the win32 build. Anyway, the `HandleInput` function looks like this:

```cpp
void GameWorld::HandleInput(CCPoint input)
{
   // don't accept input if popup is active or if player is dead
   if(is_popup_active_ || player_->is_dying_)
      return;

   CCPoint input_abs = CCPoint(fabs(input.x), fabs(input.y));

   // calculate player speed based on how much device has tilted
   // greater speed multipliers for greater tilt values
   player_->speed_.x = input.x * ( (input_abs.x > 0.3f) ? 36 : ( (input_abs.x > 0.2f) ? 28 : 20 ) );
   player_->speed_.y = input.y * ( (input_abs.y > 0.3f) ? 36 : ( (input_abs.y > 0.2f) ? 28 : 20 ) );

   // update the background
   background_->setPosition(ccp(input.x * -30, input.y * -30));
}
```

We return from this function when there is a popup active or when the player is dying. That next bit of calculation is necessary to offer a more sensitive controlling experience. Basically, the user doesn't have to tilt his device all the way to the left if he wants the player to move to the left at full speed. Thus, we multiply larger values for larger readings and smaller values for smaller readings. We finally set the player's speed to the resultant calculation. One of the cool features of an accelerometer based game is to allow the users to calibrate the accelerometer. That is left to you, my intelligent reader, as an exercise. Now that we have recorded the user's input, let's add the movement logic into the `Player` class.
Moving the player

We had called the scheduleUpdate function in the init function of the Player class. Thus, we define the update function to handle the movement and rotation of the player:

```cpp
void Player::update(float dt)
{
    CCDrawNode::update(dt);
    CCPoint previous_position = m_obPosition;
    UpdatePosition();
    UpdateRotation(previous_position);
}
```

We first save the previous position of the player because we need it while setting the rotation. The UpdateRotation function will just use ccpToAngle function to set the rotation of the player with a bit of easing, so we will skip it and discuss only the UpdatePosition function as follows:

```cpp
void Player::UpdatePosition()
{
    // don't move if speed is too low
    if(ccpLength(speed_) > 0.75f)
    {
        // add speed but limit movement within the boundary
        CCPoint next_position = ccpAdd(m_obPosition, speed_);
        if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_,
                                  next_position, PLAYER_RADIUS))
        {
            setPosition(next_position);
        }
        else
        {
            if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_, CCPoint(
                                  next_position.x - speed_.x, next_position.y), PLAYER_RADIUS))
            {
                setPosition(ccp(next_position.x - speed_.x, next_position.y));
            }
            else if(RECT_CONTAINS_CIRCLE(game_world_->boundary_rect_,
                                          CCPoint(
                                          next_position.x, next_position.y - speed_.y), PLAYER_RADIUS))
            {
                setPosition(ccp(next_position.x, next_position.y - speed_.y));
            }
        }
    }
}
```
We ignore the values that are less than a certain threshold; otherwise, the player will constantly be jerking around the screen owing to the accelerometer readings constantly fluctuating by minor values. The logic is to calculate the next position by adding up the speed and finally limiting the player within the boundary defined by GameWorld.

Adding progression and difficulty levels

By now, we have a fully functioning game but we still haven't ensured the game is challenging and addictive for the user. We must engage the player by increasing the intensity and difficulty of the game progressively. In our previous game, we had the option of designing levels that get more difficult, but this game isn't level based and is different every time.

With that in mind, a bit of spice is added by creating a variety of formations in which the enemies will spawn on screen. These formations will be increasingly difficult with the difficulty completely based on how long the user has managed to survive. We define a few enums and constants in GameGlobals.h before writing the functions that will add progression to our game:

```c
enum ESkillTimer
{
    E_SKILL1 = 10,
    E_SKILL2 = 30,
    E_SKILL3 = 45,
    E_SKILL4 = 60,
    E_SKILL5 = 90,
    E_SKILL6 = 120,
};
```

First up, we have an enum called ESkillTimer that represents the skill level in terms of the number of seconds the user has survived the game. Next, we have enum EEnemyFormation defined as follows:

```c
enum EEnemyFormation
{
    E_FORMATION_RANDOM_EASY = 0,
    E_FORMATION_VERTICAL_EASY,
    E_FORMATION_HORIZONTAL_EASY,
    E_FORMATION_POLYGON_EASY,
    E_FORMATION_RANDOM_MEDIUM,
    E_FORMATION_VERTICAL_MEDIUM,
    E_FORMATION_HORIZONTAL_MEDIUM,
    E_FORMATION_POLYGON_MEDIUM,
    E_FORMATION_RANDOM_HARD,
```
EEnemyFormation is an enum representing the various types of formations the enemies will be positioned in when they are added to the GameWorld. Next, we have a few arrays pre-defined as follows:

```c++
const int GameGlobals::skill1_formations[] = {0, 4};
const int GameGlobals::skill2_formations[] = {4, 4, 4, 1, 1, 1, 2, 2, 2};
const int GameGlobals::skill3_formations[] = {4, 4, 4, 8, 8, 1, 1, 2, 2, 5, 5, 6, 6, 6, 3, 3};
const int GameGlobals::skill4_formations[] = {4, 4, 8, 8, 5, 5, 6, 6, 6, 6, 3, 3, 7, 7, 7};
const int GameGlobals::skill5_formations[] = {8, 8, 8, 3, 3, 3, 5, 5, 6, 6, 9, 9, 10, 10, 7, 7, 7};
const int GameGlobals::skill6_formations[] = {8, 8, 8, 5, 5, 6, 6, 9, 9, 10, 10, 7, 7, 7, 11, 11, 11};
```

We also have arrays that specify the frequency of the various formations for a specific skill level. We will stitch these different bits of information together in the GetEnemyFormationType function:

```c++
EEnemyFormation GameWorld::GetEnemyFormationType()
{
    // return a formation type from a list of formation types, based on time user has been playing
    // the longer the user has survived, the more difficult the formations will be
    if(seconds_ > E_SKILL6)
    {
        int random_index = CCRANDOM_0_1() * GameGlobals::skill6_formations_size;
        return (EEnemyFormation)(GameGlobals::skill6_formations[random_index]);
    }
    else if(seconds_ > E_SKILL5)
    {
        int random_index = CCRANDOM_0_1() * GameGlobals::skill5_formations_size;
        return (EEnemyFormation)(GameGlobals::skill5_formations[random_index]);
    }
    else
    {
        // other conditions...
    }
}
```
else if(seconds_ > E_SKILL4)
{
    int random_index = CCRANDOM_0_1() * GameGlobals::skill4_formations_size;
    return (EEnemyFormation)(GameGlobals::skill4_formations[random_index]);
}
else if(seconds_ > E_SKILL3)
{
    int random_index = CCRANDOM_0_1() * GameGlobals::skill3_formations_size;
    return (EEnemyFormation)(GameGlobals::skill3_formations[random_index]);
}
else if(seconds_ > E_SKILL2)
{
    int random_index = CCRANDOM_0_1() * GameGlobals::skill2_formations_size;
    return (EEnemyFormation)(GameGlobals::skill2_formations[random_index]);
}
else if(seconds_ > E_SKILL1)
{
    int random_index = CCRANDOM_0_1() * GameGlobals::skill1_formations_size;
    return (EEnemyFormation)(GameGlobals::skill1_formations[random_index]);
}
else
{
    return E_Formation_Random_Easy;
}

This function first checks up to which skill level the user has managed to survive and then returns an appropriate formation for the respective skill level. Even though the formations are chosen randomly, we still have control over how often a given formation can pop up for a given difficulty level. All we need to do is increase or decrease the number of occurrences of a formation type within the array for a skill level.
Now that we have the type of formation based on difficulty, we can proceed by actually adding the enemies to the game in the `AddEnemyFormation` function:

```cpp
void GameWorld::AddEnemyFormation()
{
    // fetch an enemy formation
    EEnemyFormation type = GetEnemyFormationType();
    // fetch a list of positions for the given formation
    vector<CCPoint> formation = GameGlobals::GetEnemyFormation(
        type, boundary_rect_, player_->getPosition());
    int num_enemies_to_create = formation.size();
    int num_enemies_on_screen = enemies_->count();
    // limit the total number of enemies to MAX_ENEMIES
    if(num_enemies_on_screen + num_enemies_to_create >= MAX_ENEMIES)
    {
        num_enemies_to_create = MAX_ENEMIES - num_enemies_on_screen;
    }
    // create, add & position enemies based on the formation
    for(int i = 0; i < num_enemies_to_create; ++i)
    {
        Enemy* enemy = Enemy::create(this);
        enemy->setPosition(formation[i]);
        enemy->Spawn(i * ENEMY_SPAWN_DELAY);
        addChild(enemy, E_LAYER_ENEMIES);
        enemies_->addObject(enemy);
    }
}
```

This function begins by fetching the type of formation to create and passes it into the `GetEnemyFormation` function of the `GameGlobals` class. The `GetEnemyFormation` function will return a vector of `CCPoint` that we will use to position the enemies. At any given point, the total number of enemies on screen should not exceed 250, which in this case is held by constant `MAX_ENEMIES` defined in `GameGlobals.h`. We ensure the preceding condition is met and then run a loop where we create an `Enemy` object, set its position, call its `Spawn` function, and finally add it to the screen as well as the `enemies_` `CCARRAY`.

By adding progression to Inverse Universe, we complete our fourth game. I have skipped discussing some of the functionality in the chapter, such as the `BackgroundManager` class, the way enemy formations are created and some of the helper functions. You will find the code for such features and more in the source bundle. Rest assured the comments will be enough to explain what's happening.
Summary

With Inverse Universe, we went back to the drawing board and came out with flying colors! You learned the difference between CCDrawingPrimitives and CCDrawNode. We also extended the CCDrawNode class to create each and every one of our game's elements. We implemented circle-to-circle collision detection and finally understood how tilt controls or the accelerometer work in Cocos2d-x. As a good game design principle, you also learned how to add progression to a game that is not level-based, while still keeping it open-ended.

In our next chapter, we'll get acquainted with an extremely lively friend, a friend that you can keep applying forces to, and a friend that can take any shape and size that you'd like — yes it's none other than our great friend Box2D!
Where to buy this book

You can buy Cocos2d-x Game Development Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.