Chapter 5. Handling Game States

When we first start up a game, we expect to see a splash screen showing any branding for publishers and developers, followed by a loading screen as the game does its initial setup. After this, we are usually faced with a menu screen; here, we can change settings and start the game. Starting the game leads us to another loading screen, possibly followed by a cut scene, and finally, we are in the game. When we are in the game, we can pause our play (allowing us to change any settings), exit the game, restart the level, and so on. If we fail the level, we are shown either an animation or a game over screen depending on how the game is set up. All of these different sections of a game are called *Game States*. It is very important that we make the transition between these states as easy as possible.

In this chapter we will cover:

- Two different ways of handling states, starting with a really simple implementation and gradually building our framework implementation
- Implementing **Finite State Machines** (**FSM**)
- Adding states to the overall framework

A simple way for switching states

One of the simplest ways to handle states is to load everything we want at the game's initialization stage, but only draw and update the objects specific to each state. Let's look at an example of how this could work. First, we can define a set of states we are going to use:

```
enum game_states
{
  MENU = 0,
  PLAY = 1,
  GAMEOVER = 2
};
```

We can then use the `Game::init` function to create the objects:

```
// create menu objects
m_pMenuObj1 = new MenuObject();
m_pMenuObj1 = new MenuObject();

// create play objects
m_pPlayer = new Player();
m_pEnemy = new Enemy();
```

```
// create game over objects…
```

Then, set our initial state:

```
m_currentGameState = MENU;
```

Next, we can change our `update` function to only use the things we want when in a specific state:

```
void Game::update()
{
  switch(m_currentGameState)
  {
    case MENU:
      m_menuObj1->update();
      m_menuObj2->update();
      break;

    case PLAY:
      m_pPlayer->update();
      m_pEnemy->update();

    case GAMEOVER:
      // do game over stuff…
  }
}
```

The `render` function would do something similar. These functions could of course still loop through arrays and use polymorphism as we originally had done, but on a state-by-state basis. Changing states is as simple as changing the value of the `m_currentGameState` variable.

If you can see issues with this method, then it is very encouraging that you are starting to think in an object-oriented way. This way of updating states would be a bit of a nightmare to maintain and the scope for error is quite large. There are too many areas that need to be updated and changed to make this a viable solution for any game larger than a simple arcade game.

Implementing finite state machines

What we really need is the ability to define our states outside the `game` class, and have the state itself take care of what it needs to load, render, and update. For this we can create what is known as an FSM. The definition of FSM, as we will use it, is a machine that can exist in a finite number of states, can exist in only one state at a time (known as the current state), and can change from one state to another (known as a transition).

A base class for game states

Let's start our implementation by creating a base class for all of our states; create a header file called `GameState.h`:

```
#include<string>
class GameState
{
public:
  virtual void update() = 0;
  virtual void render() = 0;

  virtual bool onEnter() = 0;
  virtual bool onExit() = 0;

  virtual std::string getStateID() const = 0;
};
```

Just like our `GameObject` class, this is an abstract base class; we aren't actually putting any functionality into it, we just want all of our derived classes to follow this blueprint. The `update` and `render` functions are self-explanatory, as they will function just like the functions we created in the `Game` class. We can think of the `onEnter` and `onExit` functions as similar to other `load` and `clean` functions; we call the `onEnter` function as soon as a state is created and `onExit` once it is removed. The last function is a getter for the state ID; each state will need to define this function and return its own `staticconst` ID. The ID is used to ensure that states don't get repeated. There should be no need to change to the same state, so we check this using the state ID.

That's it for our `GameState` base class; we can now create some test states that derive from this class. We will start with a state called `MenuState`. Go ahead and create `MenuState.h` and `MenuState.cpp` in our project, open up `MenuState.h`, and start coding:

```
#include"GameState.h"
```

```
class MenuState : public GameState
{
public:

  virtual void update();
  virtual void render();

  virtual bool onEnter();
  virtual bool onExit();

  virtual std::string getStateID() const { return s_menuID; }

private:

  static const std::string s_menuID;
};
```

We can now define these methods in our `MenuState.cpp` file. We will just display some text in the console window for now while we test our implementation; we will give this state an ID of `"MENU"`:

```
#include "MenuState.h"

const std::string MenuState::s_menuID = "MENU";

void MenuState::update()
{
  // nothing for now
}

void MenuState::render()
{
  // nothing for now
}

bool MenuState::onEnter()
{
  std::cout << "entering MenuState\n";
  return true;
}

bool MenuState::onExit()
{
  std::cout << "exiting MenuState\n";
  return true;
}
```

We will now create another state called `PlayState`, create `PlayState.h` and `PlayState.cpp` in our project, and declare our methods in the header file:

```
#include "GameState.h"

class PlayState : public GameState
{
public:

  virtual void update();
  virtual void render();

  virtual bool onEnter();
  virtual bool onExit();

  virtual std::string getStateID() const { return s_playID; }

private:

  static const std::string s_playID;
};
```

This header file is the same as `MenuState.h` with the only difference being `getStateID` returning this class' specific ID (`"PLAY"`). Let's define our functions:

```
#include "PlayState.h"

const std::string PlayState::s_playID = "PLAY";

void PlayState::update()
{
  // nothing for now
}

void PlayState::render()
{
  // nothing for now
}

bool PlayState::onEnter()
{
  std::cout << "entering PlayState\n";
  return true;
}

bool PlayState::onExit()
{
  std::cout << "exiting PlayState\n";
  return true;
}
```

We now have two states ready for testing; we must next create our FSM so that we can handle them.

Implementing FSM

Our FSM is going to need to handle our states in a number of ways, which include:

- **Removing one state and adding another**: We will use this way to completely change states without leaving the option to return
- **Adding one state without removing the previous state**: This way is useful for pause menus and so on
- **Removing one state without adding another**: This way will be used to remove pause states or any other state that had been pushed on top of another one

Now that we have come up with the behavior we want our FSM to have, let's start creating the class. Create the `GameStateMachine.h` and `GameStateMachine.cpp` files in our project. We will start by declaring our functions in the header file:

```
#include "GameState.h"

class GameStateMachine
{
public:

  void pushState(GameState* pState);
  void changeState(GameState* pState);
  void popState();
};
```

We have declared the three functions we need. The `pushState` function will add a state without removing the previous state, the `changeState` function will remove the previous state before adding another, and finally, the `popState` function will remove whichever state is currently being used without adding another. We will need a place to store these states; we will use a vector:

```
private:

std::vector<GameState*> m_gameStates;
```

In the `GameStateMachine.cpp` file, we can define these functions and then go through them step-by-step:

```
void GameStateMachine::pushState(GameState *pState)
{
  m_gameStates.push_back(pState);
  m_gameStates.back()->onEnter();
}
```

This is a very straightforward function; we simply push the passed-in `pState` parameter into the `m_gameStates` array and then call its `onEnter` function:

```
void GameStateMachine::popState()
{
  if(!m_gameStates.empty())
  {
    if(m_gameStates.back()->onExit())
    {
      delete m_gamestates.back();
      m_gameStates.pop_back();
    }
  }
}
```

Another simple function is `popState`. We first check if there are actually any states available to remove, and if so, we call the `onExit` function of the current state and then remove it:

```
void GameStateMachine::changeState(GameState *pState)
{
  if(!m_gameStates.empty())
  {
    if(m_gameStates.back()->getStateID() == pState-
>getStateID())
    {
      return; // do nothing
    }

    if(m_gameStates.back()->onExit())
    {
      delete m_gamestates.back();
      m_gameStates.pop_back();
    }
  }

  // push back our new state
  m_gameStates.push_back(pState);

  // initialise it
  m_gameStates.back()->onEnter();
}
```

Our third function is a little more complicated. First, we must check if there are already any states in the array, and if there are, we check whether their state ID is the same as the current one, and if it is, then we do nothing. If the state IDs do not match, then we remove the current state, add our new `pState`, and call its `onEnter` function. Next, we will add new `GameStateMachine` as a member of the `Game` class:

```
GameStateMachine* m_pGameStateMachine;
```

We can then use the `Game::init` function to create our state machine and add our first state:

```
m_pGameStateMachine = new GameStateMachine();
m_pGameStateMachine->changeState(new MenuState());
```

The `Game::handleEvents` function will allow us to move between our states for now:

```
void Game::handleEvents()
{
  TheInputHandler::Instance()->update();

  if(TheInputHandler::Instance()-
>isKeyDown(SDL_SCANCODE_RETURN))
  {
    m_pGameStateMachine->changeState(new PlayState());
  }
}
```

When we press the *Enter* key, the state will change. Test the project and you should get the following output after changing states:

```
entering MenuState
exiting MenuState
entering PlayState
```

We now have the beginnings of our FSM and can next add `update` and `render` functions to our `GameStateMachine` header file:

```
void update();
void render();
```

We can define them in our `GameStateMachine.cpp` file:

```
void GameStateMachine::update()
{
  if(!m_gameStates.empty())
  {
    m_gameStates.back()->update();
  }
}

void GameStateMachine::render()
{
  if(!m_gameStates.empty())
  {
```

```
    m_gameStates.back()->render();
  }
}
```

These functions simply check if there are any states, and if so, they update and render the current state. You will notice that we use `back()` to get the current state; this is because we have designed our FSM to always use the state at the back of the array. We use `push_back()` when adding new states so that they get pushed to the back of the array and used immediately. Our `Game` class will now use the FSM functions in place of its own `update` and `render` functions:

```
void Game::render()
{
  SDL_RenderClear(m_pRenderer);

  m_pGameStateMachine->render();

  SDL_RenderPresent(m_pRenderer);
}

void Game::update()
{
  m_pGameStateMachine->update();
}
```

Our FSM is now in place.

Implementing menu states

We will now move on to creating a simple menu state with visuals and mouse handling. We will use two new screenshots for our buttons, which are available with the source code downloads:

The following screenshot shows the exit feature:

These are essentially sprite sheets with the three states of our button. Let's create a new class for these buttons, which we will call `MenuButton`. Go ahead and create `MenuButton.h` and `MenuButton.cpp`. We will start with the header file:

```
class MenuButton : public SDLGameObject
{
public:

  MenuButton(const LoaderParams* pParams);

  virtual void draw();
  virtual void update();
  virtual void clean();
};
```

By now this should look very familiar and it should feel straightforward to create new types. We will also define our button states as an enumerated type so that our code becomes more readable; put this in the header file under `private`:

```
enum button_state
{
  MOUSE_OUT = 0,
  MOUSE_OVER = 1,
  CLICKED = 2
};
```

Open up the `MenuButton.cpp` file and we can start to flesh out our `MenuButton` class:

```
MenuButton::MenuButton(const LoaderParams* pParams) :
SDLGameObject(pParams)
{
```

```
  m_currentFrame = MOUSE_OUT; // start at frame 0
}

void MenuButton::draw()
{
  SDLGameObject::draw(); // use the base class drawing
}

void MenuButton::update()
{
  Vector2D* pMousePos = TheInputHandler::Instance()
  ->getMousePosition();

  if(pMousePos->getX() < (m_position.getX() + m_width)
  && pMousePos->getX() > m_position.getX()
  && pMousePos->getY() < (m_position.getY() + m_height)
  && pMousePos->getY() > m_position.getY())
  {
    m_currentFrame = MOUSE_OVER;

    if(TheInputHandler::Instance()->getMouseButtonState(LEFT))
    {
      m_currentFrame = CLICKED;
    }
  }
  else
  {
    m_currentFrame = MOUSE_OUT;
  }
}

void MenuButton::clean()
{
  SDLGameObject::clean();
}
```

The only thing really new in this class is the `update` function. Next, we will go through each step of this function:

- First, we get the coordinates of the mouse pointer and store them in a pointer to a `Vector2D` object:

```
Vector2D* pMousePos = TheInputHandler::Instance()-
>getMousePosition();
```

- Now, check whether the mouse is over the button or not. We do this by first checking whether the mouse position is less than the position of the right-hand side of the button (*x position* + *width*). We then check if the mouse position is greater than the position of the left-hand side of the button (*x position*). The y-position check is essentially the same with *y position* + *height* and *y position* for bottom and top respectively:

```
if(pMousePos->getX() < (m_position.getX() + m_width)
&& pMousePos->getX() > m_position.getX()
&& pMousePos->getY() < (m_position.getY() + m_height)
&& pMousePos->getY() > m_position.getY())
```

- If the previous check is true, we know that the mouse is hovering over our button; we set its frame to `MOUSE_OVER (1)`:

```
m_currentFrame = MOUSE_OVER;
```

- We can then check whether the mouse has been clicked; if it has, then we set the current frame to `CLICKED(2)`:

```
if(TheInputHandler::Instance()-
>getMouseButtonState(LEFT))
{
  m_currentFrame = CLICKED;
}
```

- If the check is not true, then we know the mouse is outside the button and we set the frame to `MOUSE_OUT (0)`:

```
else
{
  m_currentFrame = MOUSE_OUT;
}
```

We can now test out our reusable button class. Open up our previously created `MenuState.hand`, which we will implement for real. First, we are going to need a vector of `GameObject*` to store our menu items:

```
std::vector<GameObject*> m_gameObjects;
```

Inside the `MenuState.cpp` file, we can now start handling our menu items:

```
void MenuState::update()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->update();
  }
}
void MenuState::render()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->draw();
  }
}
```

The `onExit` and `onEnter` functions can be defined as follows:

```
bool MenuState::onEnter()
{
  if(!TheTextureManager::Instance()->load("assets/button.png",
  "playbutton", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  if(!TheTextureManager::Instance()->load("assets/exit.png",
  "exitbutton", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  GameObject* button1 = new MenuButton(new LoaderParams(100,
100,
  400, 100, "playbutton"));
  GameObject* button2 = new MenuButton(new LoaderParams(100,
300,
  400, 100, "exitbutton"));

  m_gameObjects.push_back(button1);
  m_gameObjects.push_back(button2);

  std::cout << "entering MenuState\n";
  return true;
}

bool MenuState::onExit()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->clean();
  }
  m_gameObjects.clear();
  TheTextureManager::Instance()
  ->clearFromTextureMap("playbutton");
  TheTextureManager::Instance()
  ->clearFromTextureMap("exitbutton");

  std::cout << "exiting MenuState\n";
  return true;
}
```

We use `TextureManager` to load our new images and then assign these textures to two buttons. The `TextureManager` class also has a new function called `clearFromTextureMap`, which takes the ID of the texture we want to remove; it is defined as follows:

```
void TextureManager::clearFromTextureMap(std::string id)
{
  m_textureMap.erase(id);
}
```

This function enables us to clear only the textures from the current state, not the entire texture map. This is essential when we push states and then pop them, as we do not want the popped state to clear the original state's textures.

Everything else is essentially identical to how we handle objects in the `Game` class. Run the project and we will have buttons that react to mouse events. The window will look like the following screenshot (go ahead and test it out):

Function pointers and callback functions

Our buttons react to rollovers and clicks but do not actually do anything yet. What we really want to achieve is the ability to create `MenuButton` and pass in the function we want it to call once it is clicked; we can achieve this through the use of function pointers. Function pointers do exactly as they say: they point to a function. We can use classic C style function pointers for the moment, as we are only going to use functions that do not take any parameters and always have a return type of `void` (therefore, we do not need to make them generic at this point).

The syntax for a function pointer is like this:

```
returnType (*functionName)(parameters);
```

We declare our function pointer as a private member in `MenuButton.h` as follows:

```
void (*m_callback)();
```

We also add a new member variable to handle clicking better:

```
bool m_bReleased;
```

Now we can alter the constructor to allow us to pass in our function:

```
MenuButton(const LoaderParams* pParams, void (*callback)());
```

In our `MenuButton.cpp` file, we can now alter the constructor and initialize our pointer with the initialization list:

```
MenuButton::MenuButton(const LoaderParams* pParams, void
(*callback)() ) : SDLGameObject(pParams), m_callback(callback)
```

The `update` function can now call this function:

```
void MenuButton::update()
{
  Vector2D* pMousePos = TheInputHandler::Instance()
  ->getMousePosition();

  if(pMousePos->getX() < (m_position.getX() + m_width)
  && pMousePos->getX() > m_position.getX()
  && pMousePos->getY() < (m_position.getY() + m_height)
  && pMousePos->getY() > m_position.getY())
  {
    if(TheInputHandler::Instance()->getMouseButtonState(LEFT)
    && m_bReleased)
    {
      m_currentFrame = CLICKED;

      m_callback(); // call our callback function

      m_bReleased = false;
    }
    else if(!TheInputHandler::Instance()
    ->getMouseButtonState(LEFT))
    {
      m_bReleased = true;
      m_currentFrame = MOUSE_OVER;
    }
  }
  else
  {
    m_currentFrame = MOUSE_OUT;
  }
}
```

Note that this `update` function now uses the `m_bReleased` value to ensure we release the mouse button before doing the callback again; this is how we want our clicking to behave.

In our `MenuState.h` object, we can declare some functions that we will pass into the constructors of our `MenuButton` objects:

```
private:
// call back functions for menu items
static void s_menuToPlay();
static void s_exitFromMenu();
```

We have declared these functions as static; this is because our callback functionality will only support static functions. It is a little more complicated to handle regular member functions as function pointers, so we will avoid this and stick to static functions. We can define these functions in the `MenuState.cpp` file:

```
void MenuState::s_menuToPlay()
{
  std::cout << "Play button clicked\n";
}

void MenuState::s_exitFromMenu()
{
  std::cout << "Exit button clicked\n";
}
```

We can pass these functions into the constructors of our buttons:

```
GameObject* button1 = new MenuButton(new LoaderParams(100,
100, 400, 100, "playbutton"), s_menuToPlay);
GameObject* button2 = new MenuButton(new LoaderParams(100,
300, 400, 100, "exitbutton"), s_exitFromMenu);
```

Test our project and you will see our functions printing to the console. We are now passing in the function we want our button to call once it is clicked; this functionality is great for our buttons. Let's test the exit button with some real functionality:

```
void MenuState::s_exitFromMenu()
{
  TheGame::Instance()->quit();
}
```

Now clicking on our exit button will exit the game. The next step is to allow the `s_menuToPlay` function to move to `PlayState`. We first need to add a getter to the

`Game.h` file to allow us to access the state machine:

```
GameStateMachine* getStateMachine(){ return
m_pGameStateMachine; }
```

We can now use this to change states in `MenuState`:

```
void MenuState::s_menuToPlay()
{
  TheGame::Instance()->getStateMachine()->changeState(new
  PlayState());
}
```

Go ahead and test; `PlayState` does not do anything yet, but our console output should show the movement between states.

Implementing the temporary play state

We have created `MenuState`; next, we need to create `PlayState` so that we can visually see the change in our states. For `PlayState` we will create a player object that uses our `helicopter.png` image and follows the mouse around. We will start with the `Player.cpp` file and add the code to make the `Player` object follow the mouse position:

```
void Player::handleInput()
{
  Vector2D* target = TheInputHandler::Instance()
  ->getMousePosition();

  m_velocity = *target - m_position;

  m_velocity /= 50;
}
```

First, we get the current mouse location; we can then get a vector that leads from the current position to the mouse position by subtracting the current position from the mouse position. We then divide the velocity by a scalar to slow us down a little and allow us to see our helicopter catch up to the mouse rather than stick to it. Our `PlayState.h` file will now need its own vector of `GameObject*`:

```
class GameObject;

class PlayState : public GameState
{
public:
```

```
  virtual void update();
  virtual void render();

  virtual bool onEnter();
  virtual bool onExit();

  virtual std::string getStateID() const { return s_playID; }

private:

  static const std::string s_playID;

  std::vector<GameObject*> m_gameObjects;
};
```

Finally, we must update the `PlayState.cpp` implementation file to use our `Player` object:

```
const std::string PlayState::s_playID = "PLAY";

void PlayState::update()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->update();
  }
}

void PlayState::render()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->draw();
  }
}

bool PlayState::onEnter()
{
  if(!TheTextureManager::Instance()-
>load("assets/helicopter.png",
  "helicopter", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  GameObject* player = new Player(new LoaderParams(100, 100,
128,
  55, "helicopter");

  m_gameObjects.push_back(player);

  std::cout << "entering PlayState\n";
  return true;
```

```
}

bool PlayState::onExit()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->clean();
  }
  m_gameObjects.clear();
  TheTextureManager::Instance()
  ->clearFromTextureMap("helicopter");

  std::cout << "exiting PlayState\n";
  return true;
}
```

This file is very similar to the `MenuState.cpp` file, but this time we are using a `Player` object rather than the two `MenuButton` objects. There is one adjustment to our `SDLGameObject.cpp` file that will make `PlayState` look even better; we are going to flip the image file depending on the velocity of the object:

```
void SDLGameObject::draw()
{
  if(m_velocity.getX() > 0)
  {
    TextureManager::Instance()->drawFrame(m_textureID,
    (Uint32)m_position.getX(), (Uint32)m_position.getY(),
    m_width, m_height, m_currentRow, m_currentFrame,
    TheGame::Instance()->getRenderer(),SDL_FLIP_HORIZONTAL);
  }
  else
  {
    TextureManager::Instance()->drawFrame(m_textureID,
    (Uint32)m_position.getX(), (Uint32)m_position.getY(),
    m_width, m_height, m_currentRow, m_currentFrame,
    TheGame::Instance()->getRenderer());
  }
}
```

We check whether the object's velocity is more than `0` (moving to the right-hand side) and flip the image accordingly. Run our game and you will now have the ability to move between `MenuState` and `PlayState` each with their own functionality and objects. The following screenshot shows our project so far:

Pausing the game

Another very important state for our games is the pause state. Once paused, the game could have all kinds of options. Our `PauseState` class will be very similar to the `MenuState`, but with different button visuals and callbacks. Here are our two new screenshots (again available in the source code download):

The following screenshot shows the resume functionality:

Let's start by creating our `PauseState.h` file in the project:

```
class GameObject;

class PauseState : public GameState
{
public:

  virtual void update();
  virtual void render();

  virtual bool onEnter();
```

```
  virtual bool onExit();

  virtual std::string getStateID() const { return s_pauseID; }

private:

  static void s_pauseToMain();
  static void s_resumePlay();

  static const std::string s_pauseID;

  std::vector<GameObject*> m_gameObjects;
};
```

Next, create our `PauseState.cpp` file:

```
const std::string PauseState::s_pauseID = "PAUSE";

void PauseState::s_pauseToMain()
{
  TheGame::Instance()->getStateMachine()->changeState(new
  MenuState());
}

void PauseState::s_resumePlay()
{
  TheGame::Instance()->getStateMachine()->popState();
}

void PauseState::update()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->update();
  }
}

void PauseState::render()
{
  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->draw();
  }
}

bool PauseState::onEnter()
{
  if(!TheTextureManager::Instance()->load("assets/resume.png",
  "resumebutton", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  if(!TheTextureManager::Instance()->load("assets/main.png",
```

```
    "mainbutton", TheGame::Instance()->getRenderer()))
    {
      return false;
    }

    GameObject* button1 = new MenuButton(new LoaderParams(200,
  100,
    200, 80, "mainbutton"), s_pauseToMain);
    GameObject* button2 = new MenuButton(new LoaderParams(200,
  300,
    200, 80, "resumebutton"), s_resumePlay);

    m_gameObjects.push_back(button1);
    m_gameObjects.push_back(button2);

    std::cout << "entering PauseState\n";
    return true;
  }

  bool PauseState::onExit()
  {
    for(int i = 0; i < m_gameObjects.size(); i++)
    {
      m_gameObjects[i]->clean();
    }
    m_gameObjects.clear();
    TheTextureManager::Instance()
    ->clearFromTextureMap("resumebutton");
    TheTextureManager::Instance()
    ->clearFromTextureMap("mainbutton");
    // reset the mouse button states to false
    TheInputHandler::Instance()->reset();

    std::cout << "exiting PauseState\n";
    return true;
  }
```

In our `PlayState.cpp` file, we can now use our new `PauseState` class:

```
  void PlayState::update()
  {
    if(TheInputHandler::Instance()-
  >isKeyDown(SDL_SCANCODE_ESCAPE))
    {
      TheGame::Instance()->getStateMachine()->pushState(new
      PauseState());
    }

    for(int i = 0; i < m_gameObjects.size(); i++)
    {
      m_gameObjects[i]->update();
    }
  }
```

This function listens for the *Esc* key being pressed, and once it has been pressed, it then pushes a new `PauseState` class onto the state array in FSM. Remember that `pushState` does not remove the old state; it merely stops using it and uses the new state. Once we are done with the pushed state, we remove it from the state array and the game continues to use the previous state. We remove the pause state using the resume button's callback:

```
void PauseState::s_resumePlay()
{
  TheGame::Instance()->getStateMachine()->popState();
}
```

The main menu button takes us back to the main menu and completely removes any other states:

```
void PauseState::s_pauseToMain()
{
  TheGame::Instance()->getStateMachine()->changeState(new
  MenuState());
}
```

Creating the game over state

We are going to create one final state, `GameOverState`. To get to this state, we will use collision detection and a new `Enemy` object in the `PlayState` class. We will check whether the `Player` object has hit the `Enemy` object, and if so, we will change to our `GameOverState` class. Our Enemy object will use a new image `helicopter2.png`:

We will make our `Enemy` object's helicopter move up and down the screen just to keep things interesting. In our `Enemy.cpp` file, we will add this functionality:

```
Enemy::Enemy(const LoaderParams* pParams) :
SDLGameObject(pParams)
{
  m_velocity.setY(2);
  m_velocity.setX(0.001);
}

void Enemy::draw()
{
  SDLGameObject::draw();
}
```

```
void Enemy::update()
{
  m_currentFrame = int(((SDL_GetTicks() / 100) %
m_numFrames));

  if(m_position.getY() < 0)
  {
    m_velocity.setY(2);
  }
  else if(m_position.getY() > 400)
  {
    m_velocity.setY(-2);
  }

  SDLGameObject::update();
}
```

We can now add an `Enemy` object to our `PlayState` class:

```
bool PlayState::onEnter()
{
  if(!TheTextureManager::Instance()-
>load("assets/helicopter.png",
  "helicopter", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  if(!TheTextureManager::Instance()
  ->load("assets/helicopter2.png", "helicopter2",
  TheGame::Instance()->getRenderer()))
  {
    return false;
  }


  GameObject* player = new Player(new LoaderParams(500, 100,
128,
  55, "helicopter"));
  GameObject* enemy = new Enemy(new LoaderParams(100, 100,
128,
  55, "helicopter2"));

  m_gameObjects.push_back(player);
  m_gameObjects.push_back(enemy);

  std::cout << "entering PlayState\n";
  return true;
}
```

Running the game will allow us to see our two helicopters:

Before we cover collision detection, we are going to create our `GameOverState` class. We will be using two new images for this state, one for new `MenuButton` and one for a new type, which we will call `AnimatedGraphic`:

The following screenshot shows the game over functionality:

`AnimatedGraphic` is very similar to other types, so I will not go into too much detail here; however, what is important is the added value in the constructor that controls the speed of the animation, which sets the private member variable `m_animSpeed`:

```
AnimatedGraphic::AnimatedGraphic(const LoaderParams* pParams,
int animSpeed) : SDLGameObject(pParams),
m_animSpeed(animSpeed)
{

}
```

The `update` function will use this value to set the speed of the animation:

```
void AnimatedGraphic::update()
{
```

```
  m_currentFrame = int(((SDL_GetTicks() / (1000 /
m_animSpeed)) %
  m_numFrames));
}
```

Now that we have the `AnimatedGraphic` class, we can implement our `GameOverState` class. Create `GameOverState.h` and `GameOverState.cpp` in our project; the header file we will create should look very familiar, as given in the following code:

```
class GameObject;

class GameOverState : public GameState
{
public:

  virtual void update();
  virtual void render();

  virtual bool onEnter();
  virtual bool onExit();

  virtual std::string getStateID() const {return
s_gameOverID;}

private:

  static void s_gameOverToMain();
  static void s_restartPlay();

  static const std::string s_gameOverID;

  std::vector<GameObject*> m_gameObjects;
};
```

Our implementation file is also very similar to other files already covered, so again I will only cover the parts that are different. First, we define our static variables and functions:

```
const std::string GameOverState::s_gameOverID = "GAMEOVER";

void GameOverState::s_gameOverToMain()
{
  TheGame::Instance()->getStateMachine()->changeState(new
  MenuState());
}

void GameOverState::s_restartPlay()
{
  TheGame::Instance()->getStateMachine()->changeState(new
  PlayState());
}
```

The `onEnter` function will create three new objects along with their textures:

```
bool GameOverState::onEnter()
{
  if(!TheTextureManager::Instance()-
>load("assets/gameover.png",
  "gameovertext", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  if(!TheTextureManager::Instance()->load("assets/main.png",
  "mainbutton", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  if(!TheTextureManager::Instance()-
>load("assets/restart.png",
  "restartbutton", TheGame::Instance()->getRenderer()))
  {
    return false;
  }

  GameObject* gameOverText = new AnimatedGraphic(new
  LoaderParams(200, 100, 190, 30, "gameovertext", 2), 2);
  GameObject* button1 = new MenuButton(new LoaderParams(200,
200,
  200, 80, "mainbutton"), s_gameOverToMain);
  GameObject* button2 = new MenuButton(new LoaderParams(200,
300,
  200, 80, "restartbutton"), s_restartPlay);

  m_gameObjects.push_back(gameOverText);
  m_gameObjects.push_back(button1);
  m_gameObjects.push_back(button2);

  std::cout << "entering PauseState\n";
  return true;
}
```

That is pretty much it for our `GameOverState` class, but we must now create a condition that creates this state. Move to our `PlayState.h` file and we will create a new function to allow us to check for collisions:

```
bool checkCollision(SDLGameObject* p1, SDLGameObject* p2);
```

We will define this function in `PlayState.cpp`:

```
bool PlayState::checkCollision(SDLGameObject* p1,
SDLGameObject*
```

```
p2)
{
  int leftA, leftB;
  int rightA, rightB;
  int topA, topB;
  int bottomA, bottomB;

  leftA = p1->getPosition().getX();
  rightA = p1->getPosition().getX() + p1->getWidth();
  topA = p1->getPosition().getY();
  bottomA = p1->getPosition().getY() + p1->getHeight();

  //Calculate the sides of rect B
  leftB = p2->getPosition().getX();
  rightB = p2->getPosition().getX() + p2->getWidth();
  topB = p2->getPosition().getY();
  bottomB = p2->getPosition().getY() + p2->getHeight();

  //If any of the sides from A are outside of B
  if( bottomA <= topB ){return false;}
  if( topA >= bottomB ){return false; }
  if( rightA <= leftB ){return false; }
  if( leftA >= rightB ){return false;}

  return true;
}
```

This function checks for collisions between two `SDLGameObject` types. For the function to work, we need to add three new functions to our `SDLGameObject` class:

```
Vector2D& getPosition() { return m_position; }
int getWidth() { return m_width; }
int getHeight() { return m_height; }
```

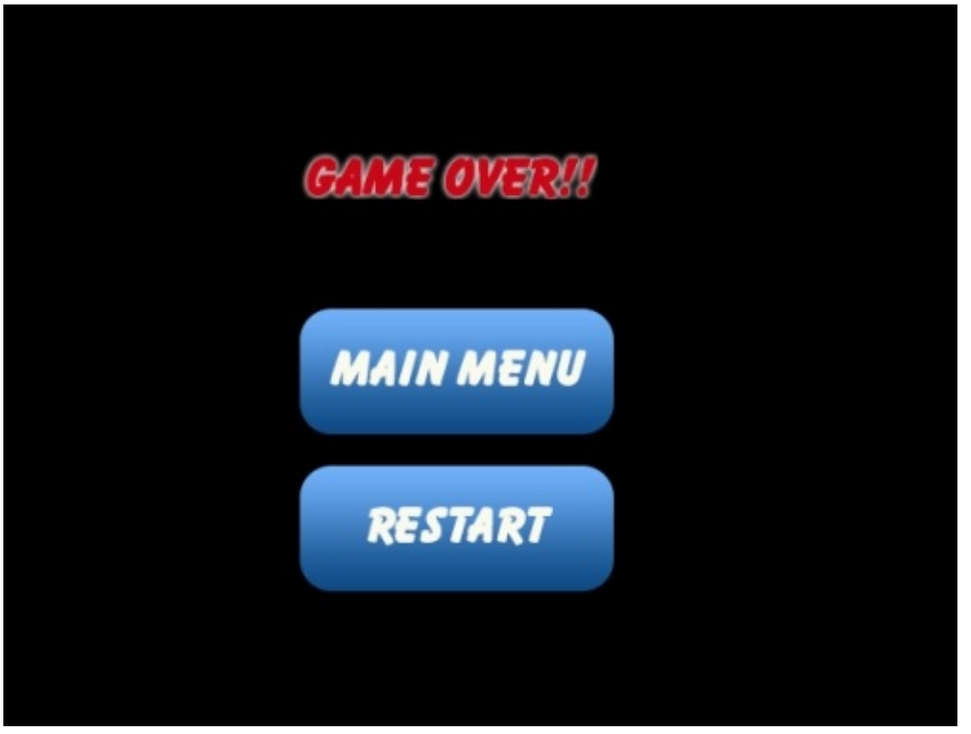
The next chapter will deal with how this function works, but for now, it is enough to know that it does. Our `PlayState` class will now utilize this collision detection in its `update` function:

```
void PlayState::update()
{
  if(TheInputHandler::Instance()-
>isKeyDown(SDL_SCANCODE_ESCAPE))
  {
    TheGame::Instance()->getStateMachine()->pushState(new
    PauseState());
  }

  for(int i = 0; i < m_gameObjects.size(); i++)
  {
    m_gameObjects[i]->update();
  }
```

```
    if(checkCollision(dynamic_cast<SDLGameObject*>
    (m_gameObjects[0]), dynamic_cast<SDLGameObject*>
    (m_gameObjects[1])))
    {
      TheGame::Instance()->getStateMachine()->pushState(new
      GameOverState());
    }
  }
```

We have to use a `dynamic_cast` object to cast our `GameObject*` class to an `SDLGameObject*` class. If `checkCollision` returns `true`, then we add the `GameOverState` class. The following screenshot shows the `GameOver` state:



Summary

This chapter has left us with something a lot more like a game than in previous chapters. We have created states for menus, pause, play, and game over with each state having its own functionality and being handled using FSM. The `Game` class now uses FSM to render and update game objects and it does not now handle objects directly, as each individual state handles its own objects. We have also created simple callback functions for our buttons using function pointers and static functions.