Week 7

Game States

Hooman Salamat

2018-03-24

Objectives

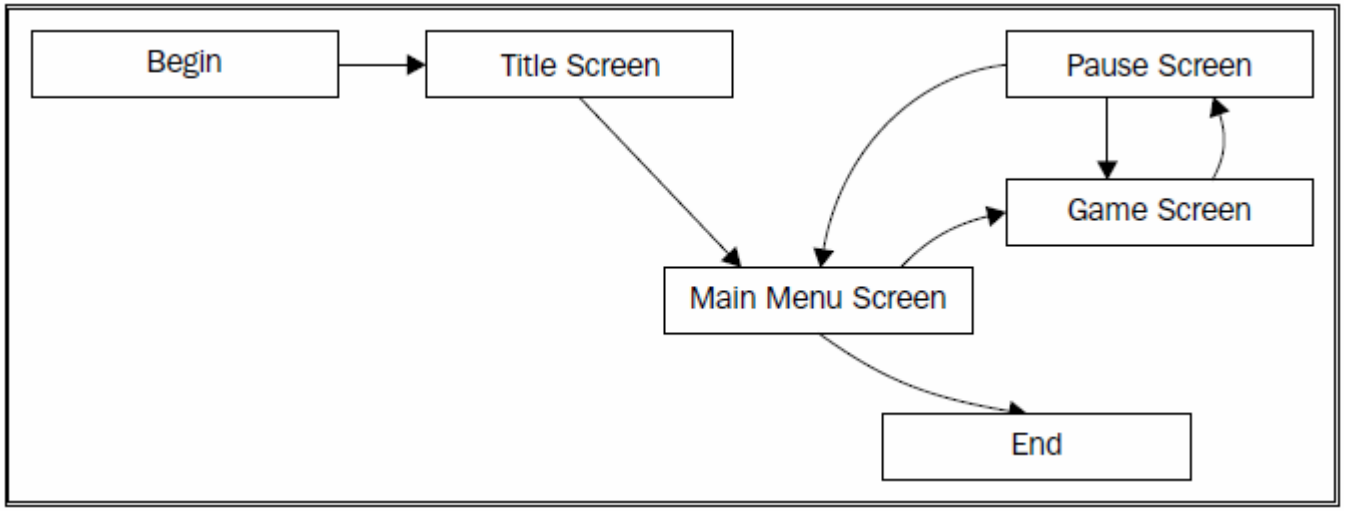
- Define states and examine the state stack
- Navigate between states
- Implement screens and menus

Defining a State

- A state could be:
  - An independent screen within the game
  - An object that encapsulates the logic and graphics of a functional group
  - Examples:
    - An introduction video
    - A title screen
    - A main menu

The State Stack

- Picture a finite state machine of all screens and how they trigger each other
- A finite state machine is a collection of states that ensures only one state is active at a time



The StateStack Class

```
class StateStack : private sf::NonCopyable
{
public:
   enum Action
   {
      Push,
      Pop,
      Clear,
   };
public:
   explicit StateStack(State::Context context);
   template <typename T>
   void registerState(States::ID stateID);
   void update(sf::Time dt);
   void draw();
   void handleEvent(const sf::Event& event);
```

The StateStack Class (cont'd.)

```
    void pushState(States::ID stateID);
    void popState();
    void clearStates();
    bool isEmpty() const;
private:
    State::Ptr createState(States::ID stateID);
    void applyPendingChanges();
    private:
    struct PendingChange
    {
        ...
        Action action;
        States::ID stateID;
    };
private:
    std::vector<State::Ptr> mStack;
    std::vector<PendingChange> mPendingList;
    State::Context mContext;
    std::map<States::ID,
    std::function<State::Ptr()>> mFactories;
};
```

The State Class

```
class State
{
public:
    typedef std::unique_ptr<State> Ptr;
    struct Context { ... };
public:
    State(StateStack& stack, Context context);
    virtual ~State();
    virtual void draw() = 0;
    virtual bool update(sf::Time dt) = 0;
    virtual bool handleEvent(const sf::Event& event) = 0;
protected:
    void requestStackPush(States::ID stateID);
    void requestStackPop();
    void requestStateClear();
    Context getContext() const;
private:
    StateStack* mStack;
    Context mContext;
};
```

The State Stack (cont'd.)

- `StateIdentifiers.hpp` contains an enum, `States`, that define unique identifiers for our game states
- We do not create all the state objects from the beginning
  - Instead, we have factory functions represented by `std::function`
  - The member variable `StateStack::mFactories` maps state IDs to those factory functions

registerState Method

- A member `StateStack::registerState()` inserts such mappings

```
template <typename T>
void StateStack::registerState(States::ID stateID)
{
  mFactories[stateID] = [this] ()
  {
    return State::Ptr(new T(*this, mContext));
  };
}
```

createState Method

- The `createState()` method takes an ID of a state, and returns a smart pointer to a newly created object of the corresponding state class

```
State::Ptr StateStack::createState(States::ID stateID)
{
   auto found = mFactories.find(stateID);
   assert(found != mFactories.end());
   return found->second();
}
```

Handling Input

```
void StateStack::handleEvent(const sf::Event& event)
{
   for (auto itr = mStack.rbegin(); itr != mStack.rend(); ++itr)
   {
      if (!(*itr)->handleEvent(event))
      return;
   }
   applyPendingChanges();
}
```

Handling Update and Draw

- The updating happens under the same guidelines of event handling
  - Both the delivery order and the stopping of update propagation to lower states
- Drawing is straightforward
  - The `StateStack` class will order every active state to render itself

Handling Update and Draw

- The updating happens under the same guidelines of event handling
  - Both the delivery order and the stopping of update propagation to lower states
- Drawing is straightforward
  - The `StateStack` class will order every active state to render itself

Delayed Pop/Push

- The `StateStack` class provides the `pushState()` and `popState()` functions to let us add and remove states from the active stack

- A special kind of pop operation is also provided, allowing a state to call `requestStackClear()`
  - Completely empties the active stack
  - These delayed processing operations are done in the function on the next slide

applyPendingChanges

```
void StateStack::applyPendingChanges()
{
    FOREACH(PendingChange change, mPendingList)
    {
        switch (change.action)
        {
            case Push:
              mStack.push_back(createState(change.stateID));
              break;
            case Pop:
              mStack.pop_back();
              break;
            case Clear:
              mStack.clear();
              break;
        }
    }
    mPendingList.clear();
}
```

The State Context

- Every screen will need to display some text or sprites among other common things
- To avoid unnecessary memory wasting by loading the same texture or font in multiple places, we have the `State::Context` structure

```
struct Context
{
   Context(sf::RenderWindow& window, TextureHolder&
     textures, FontHolder& fonts, Player& player);


   sf::RenderWindow* window;
   TextureHolder* textures;
   FontHolder* fonts;
   Player* player;
};
```

The State Context

- Every screen will need to display some text or sprites among other common things
- To avoid unnecessary memory wasting by loading the same texture or font in multiple places, we have the `State::Context` structure

```
struct Context
{
   Context(sf::RenderWindow& window, TextureHolder&
     textures, FontHolder& fonts, Player& player);


   sf::RenderWindow* window;
   TextureHolder* textures;
   FontHolder* fonts;
   Player* player;
};
```

Application Class

- Since we have now more states than the game itself, we create a new class `Application` that controls input, logic updates, and rendering
- First, we add the `mStateStack` member variable to Application
  - We register all the states in one method:

```
void Application::registerStates()
{
   mStateStack.registerState<TitleState>(States::Title);
   mStateStack.registerState<MenuState>(States::Menu);
   mStateStack.registerState<GameState>(States::Game);
   mStateStack.registerState<PauseState>(States::Pause);
}
```

Application Class (cont'd.)

- There are a few more things we must care about for a full integration of our state architecture:
- Feeding it with events in the Application::processInput() function:

```
while (mWindow.pollEvent(event))
{
   mStateStack.handleEvent(event);
}
```

- Updating with the elapsed time:

```
void Application::update(sf::Time dt)
{
   mStateStack.update(dt);
}
```

- Rendering of the stack, in the middle of the frame draw:

```
mStateStack.draw();
```

- Closing the game when no more states are left:

```
if (mStateStack.isEmpty())
   mWindow.close();
```

The Game State

```
class GameState : public State
{
public:
   GameState(StateStack& stack, Context context);
   virtual void draw();
   virtual bool update(sf::Time dt);
   virtual bool handleEvent(const sf::Event& event);
private:
   World mWorld;
   Player& mPlayer;
};
```

The Title Screen

```
class TitleState : public State
{
public:
   TitleState(StateStack& stack, Context context);
   virtual void draw();
   virtual bool update(sf::Time dt);
   virtual bool handleEvent(const sf::Event& event);
private:
   sf::Sprite mBackgroundSprite;
   sf::Text mText;
   bool mShowText;
   sf::Time mTextEffectTime;
};
```

The Title Screen (cont'd.)

- Here's how we detect the key stroke and use our state system to trigger a new state:

```
bool TitleState::handleEvent(const sf::Event& event)
{
   if (event.type == sf::Event::KeyPressed)
   {
      requestStackPop();
      requestStackPush(States::Menu);
   }
   return true;
}
```

The Title Screen (cont'd.)

- The background is merely an image covering the whole window and the blinking effect on the `sf::Text` object is achieved through this:

```
bool TitleState::update(sf::Time dt)
{
   mTextEffectTime += dt;
   if (mTextEffectTime >= sf::seconds(0.5f))
   {
      mShowText = !mShowText;
      mTextEffectTime = sf::Time::Zero;
   }
   return true;
}
```

The Main Menu

- This state is not so different from the title screen but it does implement the option selection:

```
enum OptionNames
{
   Play,
   Exit,
};

std::vector<sf::Text> mOptions;
std::size_t mOptionIndex;
```

The Main Menu (cont'd.)

- First we declare the containers of our options in the `MenuState` class
- Then, we setup and push to the `mOptions` array the `sf::Text` objects, in the constructor:

```
sf::Text playOption;
playOption.setFont(font);
playOption.setString("Play");
centerOrigin(playOption);
playOption.setPosition(context.window->getView().getSize() / 2.f);
mOptions.push_back(playOption);
```

The Main Menu (cont'd.)

- Finally, we define the most important function that helps controlling this menu:

```
void MenuState::updateOptionText()
{
   if (mOptions.empty())
     return;
   // White all texts
   FOREACH(sf::Text& text, mOptions)
     text.setColor(sf::Color::White);
   // Red the selected text
   mOptions[mOptionIndex].setColor(sf::Color::Red);
}
```

Pausing the Game

- The pause menu is a state that is not meant to work by itself but rather on top of the state stack
  - It works simultaneously with `GameState`:

```
void PauseState::draw()
{
   sf::RenderWindow& window = *getContext().window;
   window.setView(window.getDefaultView());
   sf::RectangleShape backgroundShape;
   backgroundShape.setFillColor(sf::Color(0, 0, 0, 150));
   backgroundShape.setSize(sf::Vector2f(window.getSize()));
   window.draw(backgroundShape);
   window.draw(mPausedText);
   window.draw(mInstructionText);
}
```

Pausing the Game (cont'd.)

- We also return to the main menu when *Backspace* is pressed:

```
if (event.key.code == sf::Keyboard::BackSpace)
{
  requestStateClear();
  requestStackPush(States::Menu);
}
```

- We call `requestStateClear()` instead of pop because the `PauseState` doesn't now how many states are in the stack

Loading Screen

- The example game doesn't utilize a loading screen
- Though you can have a look at a sample screen in the `LoadingState` and `ParallelTask` classes in the source