Game Engine Development II

Week 8

Hooman Salamat

Menus

Objectives

- Implement screens and menus
- Design a UI component hierarchy
- Implement containers, labels and buttons

GUI

- In most cases, the mouse will be used as an input source
  - Triggers button clicks or mouse-overs

- In the ongoing example, the keyboard is used to change menu options
- A GUI framework will be created and the first step is to reserve/create a namespace called `GUI`
- In this namespace we are defining a class called `Component` as shown on the next slide

Dangling pointer

- Problem with dangling pointer
- ref will point to undefined data!

```
int* ptr = new int(10);
int* ref = ptr;
delete ptr;
```

shared_ptr & weak_ptr

- std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object.
- std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr.

```
// empty definition
std::shared_ptr<int> sptr;

// takes ownership of pointer
sptr.reset(new int);
*sptr = 10;

// get pointer to data without taking ownership
std::weak_ptr<int> weak1 = sptr;
```

lock()

- shared_ptr : holds the real object.
- weak_ptr : uses lock to connect to the real owner or returns a NULL shared_ptr otherwise.
- weak_ptr role is similar to the role of real estate agent.
- Without agents, to buy a house, we may have to check random houses in the city.
- The agents make sure that we visit only those houses which are still accessible and available to buy.

```
// deletes managed object, acquires new pointer
    sptr.reset(new int);
    *sptr = 5;

    // get pointer to new data without taking ownership
    std::weak_ptr<int> weak2 = sptr;

    // weak1 is expired!
    if (auto tmp = weak1.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "weak1 is expired\n";

    // weak2 points to new data (5)
    if (auto tmp = weak2.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "weak2 is expired\n";
```

shared_ptr use case

- Suppose you have Game and Scene objects.
- The Game object will have pointers to its Scene objects.
- And it's likely that the Scene objects will also have a back pointer to their Game object.
- Then you have a dependency cycle. If you use shared_ptr, objects will no longer be automatically freed when you abandon reference on them, because they reference each other in a cyclic way.
- This is a memory leak.
- You break this by using weak_ptr.
- The "owner" typically use shared_ptr
- and the "owned" use a weak_ptr to its parent,
- and convert it temporarily to shared_ptr when it needs access to its parent.

```
std::shared_ptr<Parent> parentSharedPtr;

std::weak_ptr<Parent> parentWeakPtr =
parentSharedPtr; // automatic conversion to
weak from shared

std::shared_ptr<Parent> tempParentSharedPtr =
parentWeakPtr.lock(); // on the stack, from
the weak ptr

if (!tempParentSharedPtr) {
// yes, it may fail if the parent was freed
since we stored weak_ptr
    }
    else {
        // do stuff
    }

// tempParentSharedPtr is released when it
goes out of scope
```

weak_ptr

```cpp
// In this example we use shared_ptr in cyclically referenced classes. When the classes go out of scope they are NOT destroyed.
//Memory Leak Demo

#include<iostream>
#include<memory>
using namespace std;

class B;

class A
{
public:
    shared_ptr<B>bptr;
    A() {
        cout << "A created" << endl;
    }
    ~A() {
        cout << "A destroyed" << endl;
    }
};

class B
{
public:
    shared_ptr<A>aptr;
    B() {
        cout << "B created" << endl;
    }
    ~B() {
        cout << "B destroyed" << endl;
    }
};

int main()
{
    {
        shared_ptr<A> a = make_shared<A>();
        shared_ptr<B> b = make_shared<B>();
        a->bptr = b;
        b->aptr = a;
    }
}
```

```cpp
// In this example we use weak_ptr to avoid memory leak. When the classes go out of scope they are  destroyed.


#include<iostream>
#include<memory>
using namespace std;

class B;

class A
{
public:
    weak_ptr<B>bptr;
    A() {
        cout << "A created" << endl;
    }
    ~A() {
        cout << "A destroyed" << endl;
    }
};

class B
{
public:
    weak_ptr<A>aptr;
    B() {
        cout << "B created" << endl;
    }
    ~B() {
        cout << "B destroyed" << endl;
    }
};

int main()
{
    {
        shared_ptr<A> a = make_shared<A>();
        shared_ptr<B> b = make_shared<B>();
        a->bptr = b;
        b->aptr = a;
    }

    //it's going out of scope here, and hence destructed
}
```

Component Class

```
namespace GUI
{
    class Component : public sf::Drawable
                    , public sf::Transformable
                    , private sf::NonCopyable
    {
    public:
       typedef std::shared_ptr<Component> Ptr;
    public:
        Component();
        virtual ~Component();
        virtual bool isSelectable() const = 0;
        bool isSelected() const;
        virtual void select();
        virtual void deselect();
        virtual bool isActive() const;
        virtual void activate();
        virtual void deactivate();
        virtual void handleEvent(const sf::Event& event) = 0;
    private:
        bool mIsSelected;
        bool mIsActive;
    };
}
```

Other Classes

- Other classes we define:
  - GUI::Container
  - GUI::Button
  - GUI::Label

- You might recognize some of these as very common GUI types
- They are the most basic components that you will need
  - We can expand the system with more components later

Container Class

```
Container::Container() : mChildren(), mSelectedChild(-1)
{ }

void Container::pack(Component::Ptr component)
{
   mChildren.push_back(component);
   if (!hasSelection() && component->isSelectable())
       select(mChildren.size() - 1);
}

bool Container::isSelectable() const
{
   return false;
}
```

Container Class

```
void Container::handleEvent(const sf::Event& event)
{
      if (hasSelection() && mChildren[mSelectedChild]->isActive())
      {
          mChildren[mSelectedChild]->handleEvent(event);
      }
      else if (event.type == sf::Event::KeyReleased)
      {
            if (event.key.code == sf::Keyboard::W || event.key.code == sf::Keyboard::Up)
            {
               selectPrevious();
            }
            else if (event.key.code == sf::Keyboard::S || event.key.code == sf::Keyboard::Down)
            {
               selectNext();
            }
            else if (event.key.code == sf::Keyboard::Return || event.key.code == sf::Keyboard::Space)
            {
               if (hasSelection())
                        mChildren[mSelectedChild]->activate();
            }
      }
}
```

Container Class

```
void Container::select(std::size_t index)
{
     if (mChildren[index]->isSelectable())
     {
          if (hasSelection())
          mChildren[mSelectedChild]->deselect();
          mChildren[index]->select();
          mSelectedChild = index;
     }
}
void Container::selectNext()
{
     if (!hasSelection())
        return;
     // Search next component that is selectable
     int next = mSelectedChild;
     do
        next = (next + 1) % mChildren.size();
     while (!mChildren[next]->isSelectable());
     // Select that component
     select(next);
}
```

Container Class

```
void Container::selectPrevious()
{
    if (!hasSelection())
      return;
    // Search previous component that is selectable
    int prev = mSelectedChild;
    do
      prev = (prev + mChildren.size() - 1) % mChildren.size();
    while (!mChildren[prev]->isSelectable());
    // Select that component
    select(prev);
}
```

Label Class

```
Label::Label(const std::string& text, const FontHolder& fonts)
: mText(text, fonts.get(Fonts::Label), 16)
{
}

bool Label::isSelectable() const
{
    return false;
}

void Label::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    states.transform *= getTransform();
    target.draw(mText, states);
}

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

Button Class

```
Button::Button(const FontHolder& fonts, const TextureHolder& textures)
// ...
{
    mSprite.setTexture(mNormalTexture);
    mText.setPosition(sf::Vector2f(mNormalTexture.getSize() / 2u));
}

bool Button::isSelectable() const
{
    return true;
}

void Button::select()
{
    Component::select();
    mSprite.setTexture(mSelectedTexture);
}

void Button::deselect()
{
    Component::deselect();
    mSprite.setTexture(mNormalTexture);
```

Button Class (cont'd.)

```
void Button::activate()
{
     Component::activate();
     if (mIsToggle)
        mSprite.setTexture(mPressedTexture);
     if (mCallback)
        mCallback();
     if (!mIsToggle)
        deactivate();
}

void Button::deactivate()
{
    Component::deactivate();
     if (mIsToggle)
     {
          if (isSelected())
            mSprite.setTexture(mSelectedTexture);
          else
            mSprite.setTexture(mNormalTexture);
     }
}
```

Updating the Menu

```
MenuState::MenuState(StateStack& stack, Context context)
: State(stack, context), mGUIContainer()
{
    ...
    auto playButton = std::make_shared<GUI::Button>(
      *context.fonts, *context.textures);
    playButton->setPosition(100, 250);
    playButton->setText("Play");
    playButton->setCallback([this] ()
    {
      requestStackPop();
      requestStackPush(States::Game);
    });
    mGUIContainer.pack(playButton);
}
```

Updating the Menu

```
void MenuState::draw()
{
   sf::RenderWindow& window = *getContext().window;
   window.setView(window.getDefaultView());
   window.draw(mBackgroundSprite);
   window.draw(mGUIContainer);
}

bool MenuState::update(sf::Time)
{
   return true;
}

bool MenuState::handleEvent(const sf::Event& event)
{
   mGUIContainer.handleEvent(event);
   return false;
}
```

SettingsState

```
SettingsState::SettingsState(StateStack& stack, Context context)
: State(stack, context)
, mGUIContainer()
{
      mBackgroundSprite.setTexture(
          context.textures->get(Textures::TitleScreen));
      mBindingButtons[Player::MoveLeft] =
          std::make_shared<GUI::Button>(...);
      mBindingLabels[Player::MoveLeft] =
          std::make_shared<GUI::Label>(...);
      ... // More buttons and labels
      updateLabels();
      auto backButton = std::make_shared<GUI::Button>(...);
      backButton->setPosition(100, 375);
      backButton->setText("Back");
      backButton->setCallback([this] ()
      {
          requestStackPop();
      });
      mGUIContainer.pack(mBindingButtons[Player::MoveLeft]);
      mGUIContainer.pack(mBindingLabels[Player::MoveLeft]);
      ...
      mGUIContainer.pack(backButton);
}
```

SettingsState

```
void SettingsState::updateLabels()
{
	Player& player = *getContext().player;
	for (std::size_t i = 0; i < Player::ActionCount; ++i)
	{
		sf::Keyboard::Key key =
		   player.getAssignedKey(static_cast<Player::Action>(i));
		mBindingLabels[i]->setText(toString(key));
	}
}
```

SettingsState

```
bool SettingsState::handleEvent(const sf::Event& event)
{
     bool isKeyBinding = false;
     for (std::size_t action = 0; action < Player::ActionCount; ++action)
     {
          if (mBindingButtons[action]->isActive())
          {
               isKeyBinding = true;
               if (event.type == sf::Event::KeyReleased)
               {
                  getContext().player->assignKey (static_cast<Player::Action>(action),
                    event.key.code);
                  mBindingButtons[action]->deactivate();
               }
               break;
        }
     }
     if (isKeyBinding)
        updateLabels();
     else
        mGUIContainer.handleEvent(event);
     return false;
}
```