Tutorial.md

Part 1 - Introduction

Welcome to Tedm, a C++ game library which aims to help developers design simple C++ games that work across multiple operating systems. Unlike other libraries it is designed to be lightweight and take a lot of the complexity out of game design. It is solely reliant on SDL for the graphics design - however this is customizable as the library is only interactable through a series of interfaces. This library is heavily based on the design flow of https://phaser.io

Requirements

Clone the repository from here and the resources that exist in that repository. The code for this tutorial exists inside of the demos folder.

In order to follow this tutorial you'll need a very basic understanding of C++. Some additional resources that might help you understand the library would include these tutorials on SDL:

- http://lazyfoo.net/SDL_tutorials/
- https://wiki.libsdl.org/Tutorials

Once you've downloaded everything and have a copy of the repository, you'll need to make sure you have a compatible system with the code. This system has only been tested on Ubuntu 16.10 and MAC OS X. You'll need to make sure you have a recent copy of cmake as well. In order to build for Ubuntu 16.10:

```
$ sudo bash ./install-deps.sh
$ mkdir build && cd build
$ cmake .. && make
```

To build for MAX OS X

```
$ brew install sdl2
$ mkdir build && cd build
$ cmake .. && make
```

Open the part1 tutorial and you'll see a basic game state which gives you a sense of boilerplate Tedm code.

```
#include <object/State.h>
#include "Game.h"
#include <memory>

class GameState : public State {
        bool init() override { return true; }
        void destroy() override {}
        void paused() override {}
        void resumed() override {}
        void update() override {}
        void render override {}
}

int main(int argc, char*argv[]) {
        Context context;
    Game g = Game(context);
    context.width = 800;
    context.height = 600;
    GameState game_state(g);
    g.registerState("Start", make_shared<GameState>(game_state));
    g.mainLoop();
}
```

Initializing the Game g object is where you bring the entire Tedm instance to life. This is where the majority of the business logic occurs as well as where many of the events are called. By extending the interface "State", we define a new state that defines a series of functions that describe a given game state. The Context defines a persistent series of variables that persist across instances. The context by default will contain values such as the width and height of the window. If you need more information to persist across states then extend the Context object and add fields.

Each state is specified by a string name value that indicates what it should be referred to without passing around the instance of the state itself. By adding additional states you can create a series of different contexts for your game and encapsulate different behaviors, such as a start menu or a chess match. Each state is given the parent game object as a reference which contains a transition method that allows the user to switch out of the given state, like so:

```
...
```

```
game.transition("StartMenu");
...
```

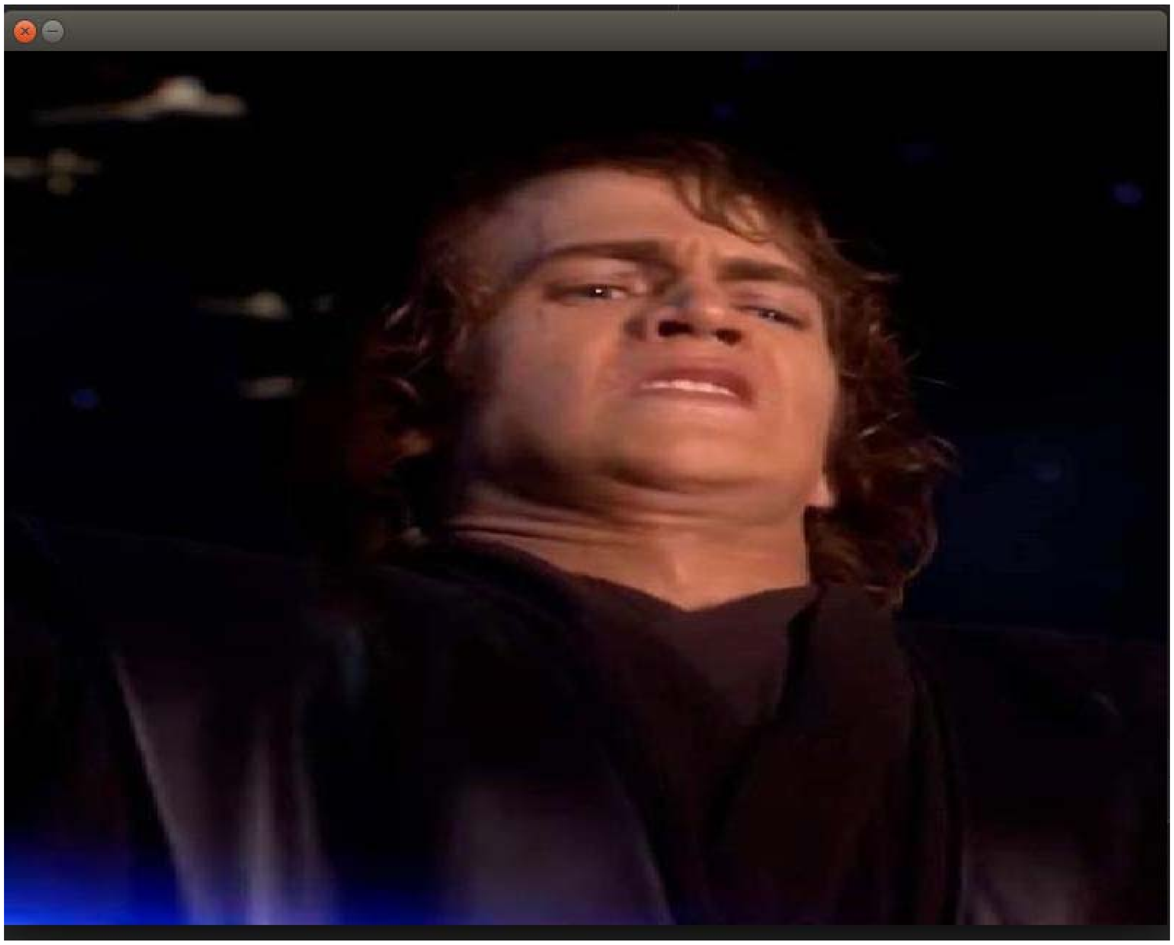
These are the basics for creating a game for Tedm. In later sections we will go over more advanced behavior for the game library.

Part 2 - Loading Assets

Let's load the assets we need for our game. You do this by creating objects inside of the init function of a state. Tedm will automatically pick up these references and read in their textures. You then will be able to display them to the screen by making a call to render.

```
#include <State.h>
#include "Game.h"
#include <memory>

class GameState : public Tedm::State {
public:
    SDL_Texture * background;

        GameState(Tedm::Game &g) : State(g, "GameState") {}

        bool init() override {
        background = graphics.add_background("../resources/dat_anakin.jpg");
        return true;
    }
        void destroy() override {}
        void paused() override {}
        void resumed() override {}
        void update() override {}
        void render() override {
        graphics.draw(background);
        }
};

int main(int argc, char*argv[]) {
        Tedm::Context context;
        Tedm::Game g = Tedm::Game(context);
        context.width = 800;
        context.height = 600;
        GameState game_state(g);
        g.registerState("Start", std::make_shared<GameState>(game_state));
    g.setStartState("Start");
        g.mainLoop();
}
```

We added several key elements here. First of all we added a background object that refers to the dat_anakin.jpg file. This will form the background of the screen.



```
#include <State.h>
#include "Game.h"
#include <memory>

class GameState : public Tedm::State {
public:
    Tedm::Object blaster;
    SDL_Texture *background;

    GameState(Tedm::Game &g) : State(g, "GameState"), blaster(graphics, "../resources/dat_anakin.png", 475, 375, 100, 100) {}

    bool init() override {
        background = graphics.add_background("../resources/dat_anakin.jpg");
        return true;
    }
    void destroy() override {}
    void paused() override {}
    void resumed() override {}
    void update() override {}
    void render() override {
        graphics.draw(background);
        blaster.draw();
    }
};

int main(int argc, char*argv[]) {
    Tedm::Context context;
    Tedm::Game g = Tedm::Game(context);
    context.width = 800;
    context.height = 600;
    GameState game_state(g);
    g.registerState("Start", std::make_shared<GameState>(game_state));
    g.setStartState("Start");
    g.mainLoop();
}
```

We added several key elements here. First of all we pointed the Object blaster to the resource "blaster.png". This will give it a reference to a texture that the graphics can draw. Then we made sure to call the draw function of the object we want shown onto the screen. If the state were to exit we would lose the state and the objects would no longer be drawn to the screen.

Part 3 - Events

One of the best features of Tedm is the functionality for a multitude of different event types. For example take this Quit_Listener:

```
class Quit_Listener : public EventListener {
    bool &isRunning;
public:
    Quit_Listener(bool &b) : isRunning(b) {
    }

    void operator()() override {
        isRunning = false;
    }
};

...
game.eventHandler.addExitListener(std::make_shared(Quit_Listener(game.isRunning)));
...
```

The convenient part of this is that we can define any kind of event we might need that are supported by hardware triggers. The event system is fairly robust in SDL and therefore the EventHandler can handle a robust set of triggers. You can also define your own custom events:

```
class Quit_Trigger : public EventTrigger {
    bool &isRunning;
public:
    Quit_Trigger(bool &b) : isRunning(b) {}

    bool triggered() override {
        return b;
    }

    void operator()() override {
        isRunning = false;
    }
};

...
game.eventHandler.addEventTrigger(std::make_shared(Quit_Trigger(game.isRunning)));
...
```