CMP105 Games Programming

Inheritance and Sprites

This week

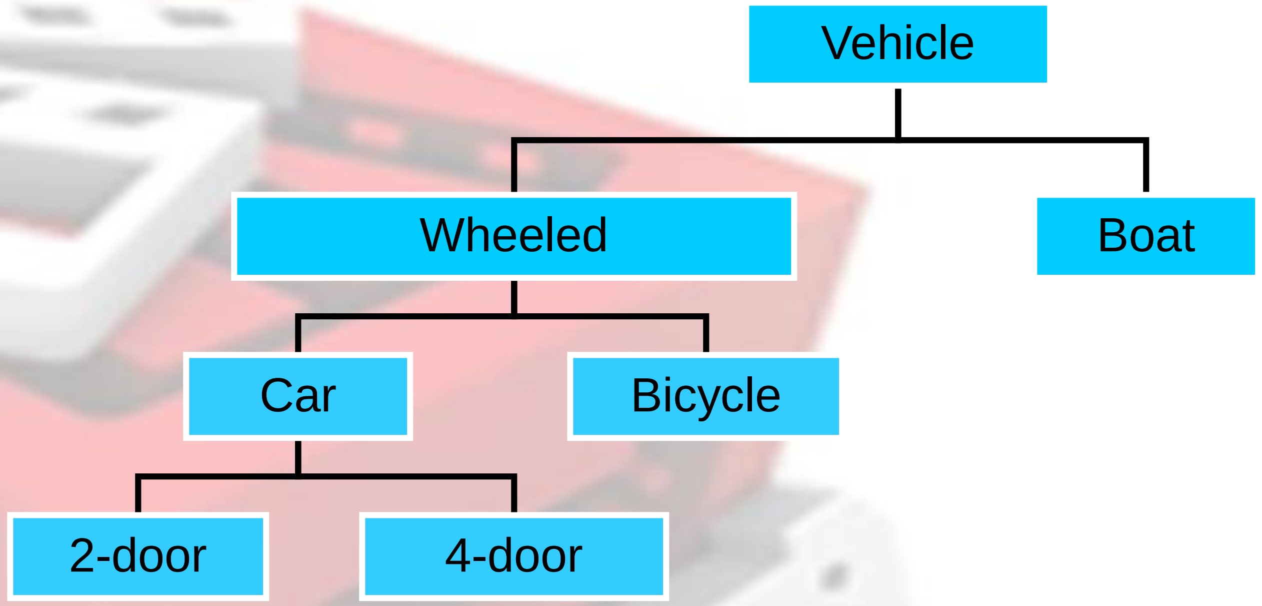
- Classes
- Inheritance
- Polymorphism
- Sprites

Classes

- A class is a model that represents the objects that make up your application.
- Think of objects as the building blocks of your applications.
- It is the interaction between these objects that create our application.
- Objects are defined by their data members and methods that provide functionality on that data.

Inheritance Hierarchy

- Higher levels are more general
- Lower levels are more specific
- Lower levels inherit properties from higher levels



C++ and Inheritance

- The mechanism by which one class acquires the properties (data and methods) of another class.

- Base Class (or superclass):
  - the general class being inherited from

- Derived Class (or subclass):
  - the specific class that inherits

Advantages of inheritance

- When a class inherits from another class, there are three benefits:

1. You can *reuse* the methods and data of the base class.
2. You can *extend* the base class by adding new data and new methods.
3. You can *modify* the base class by overloading its methods with new implementations.

Deriving one class from another

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

Class hierarchy

- A derived class can also serve as a base class for new classes.
- There is no limit on the depth of inheritance allowed in C++ but there may be performance issues.
- A class can be a base class for more than one derived class
- A class can be derived from more than one base class (multiple inheritance).

Inheritance and Containment

- Inheritance models (Application of "is a")
  - car "is a" vehicle
    - Car class inherited from vehicle class
    - Becoming more specific
- Containment models (Application of "has a")
  - Car "has a" horn
    - Car class contains a horn class
- It's easy to get a tangled mess of inheritance and containment.
  - Think carefully when using inheritance and containment!

Protected class members

- Derived classes **cannot** access the private data of the base class

- Declaring methods and data of the base class as *protected* (instead of private) allows derived classes to access them.

Constructors and Destructors

- The Base class constructor is called first then the derived class constructor.
  - Polygon::Polygon() then Rectangle::Rectangle() then class derived from rectangle

- When class goes out of scope, destructors are called in <u>reverse order</u> from the constructors.
  - The Base class destructor is called first then the derived class destructor.
    - Derived class then Rectangle::~Rectangle() then Polygon::~Polygon()

More on Constructors

- The derived class can tell the base class which constructor to use.

```
Player::Player(const float x, const float y):
GameObject(x, y)
{
   num_lives_ = 3;
   on_the_ground_ = true;
}
```

- This is an overloaded constructor which uses the overloaded constructor of the base class passing up the parameters x and y to the base class constructor.

Public, Protected and Private Inheritance

- You can control the level of access to inherited member variables and functions by using the access specifier

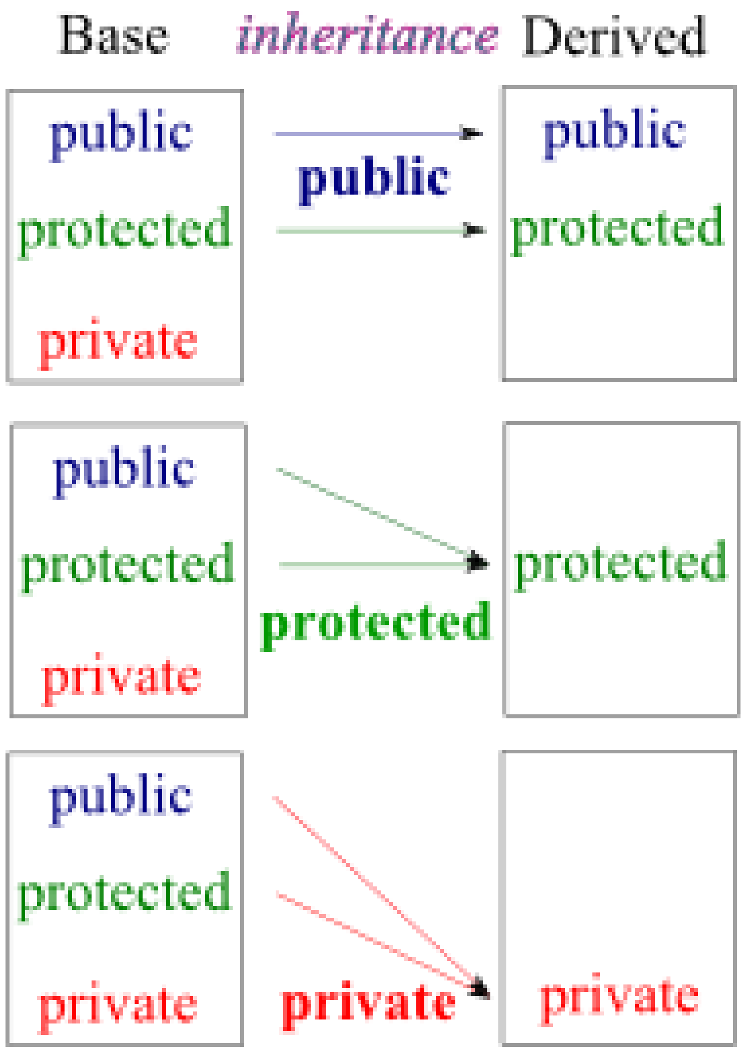
- E.g.

```
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };
```

Public, Protected and Private Inheritance

- The access specifier is most often public to allow for full use of the features inheritance.
- The only time you may wish it be protected or private is if the base class is only used to provide functionality to implement to the derived class, but not allow other objects to access to public base class methods and member variables.

Public, Protected and Private Inheritance



Polymorphism

- Any code you write to manipulate a base class <u>will also work</u> with any class derived from the base class.

- C++ general rule for passing objects to a function:

  "the actual parameters and their corresponding formal parameters must be of the same type"

- With inheritance, C++ relaxes this rule:

  "the type of the actual parameter can be a class derived from the class of the formal parameter"

Polymorphism An Example

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area ()
      { return 0; }
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

Polymorphism An Example

```
bool doSomething(Polygon* p1, Polygon p2)
{
     ...
}
```

- Any object of a class derived from Polygon can be passed to the function.
  - Rectangle and Triangle
- No need to write a new function for Rectangle and Triangle classes

Static and Dynamic Binding

```
bool doSomething(Polygon* p1, Polygon p2)
{
      ...
      int x = p1->area();
}
```

- It is possible to pass a Rectangle and a Triangle into doSomething()
- There are two Area() functions
  - Rectangle::area()
  - Triangle::area()
- which one gets called?
- Static Binding: The method to call is defined at compile time – this is what “normally” happens.
- Dynamic Binding: The method to call is defined at run time this is “polymorphism” in action.

Virtual Functions

- C++ uses *virtual functions* to implement run-time or Dynamic binding.

- To tell the compiler to generate code that manages dynamic binding, the key word *virtual* should appear before the method declaration.

Virtual Functions Example

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
    virtual int area ()
      { return 0; }
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

Pure Virtual Functions

- We can using virtual functions to define purely an interface for derived functions to adhere to without a base implementation.
- This is useful for when you are designing a class hierarchy and it doesn’t make sense to have a base implementation of a function.

Pure Virtual Function Example

```
class MeshBuilder
{
   virtual void Build() = 0;
}

class CubeBuilder : public MeshBuilder
{
   void Build(){
      // create 8 vertices and 12 triangles
   }
}

class PyramidBuilder : public MeshBuilder
{
   void Build(){
      // create 4 vertices and 4 triangles
   }
}
```

Sprites

- Sprite definition
  - A sprite is a 2D image or part of a 2D image that has been rendered to the screen
  - Written into the frame buffer pixel by pixel
  - Scaling and rotating images can be troublesome
  - A sprite requires that a graphics resource (bitmap, jpg, png) is loaded into memory

Building a Sprite class

- Inherit from sf::RectangleShape
  - Gain many of the abilities of RectangeShape
  - Can add our own
- Sprite class will be a base class for more specific derived classes
  - Player, Enemy, Level tile, Pick up, Bullet etc

Sprite.h

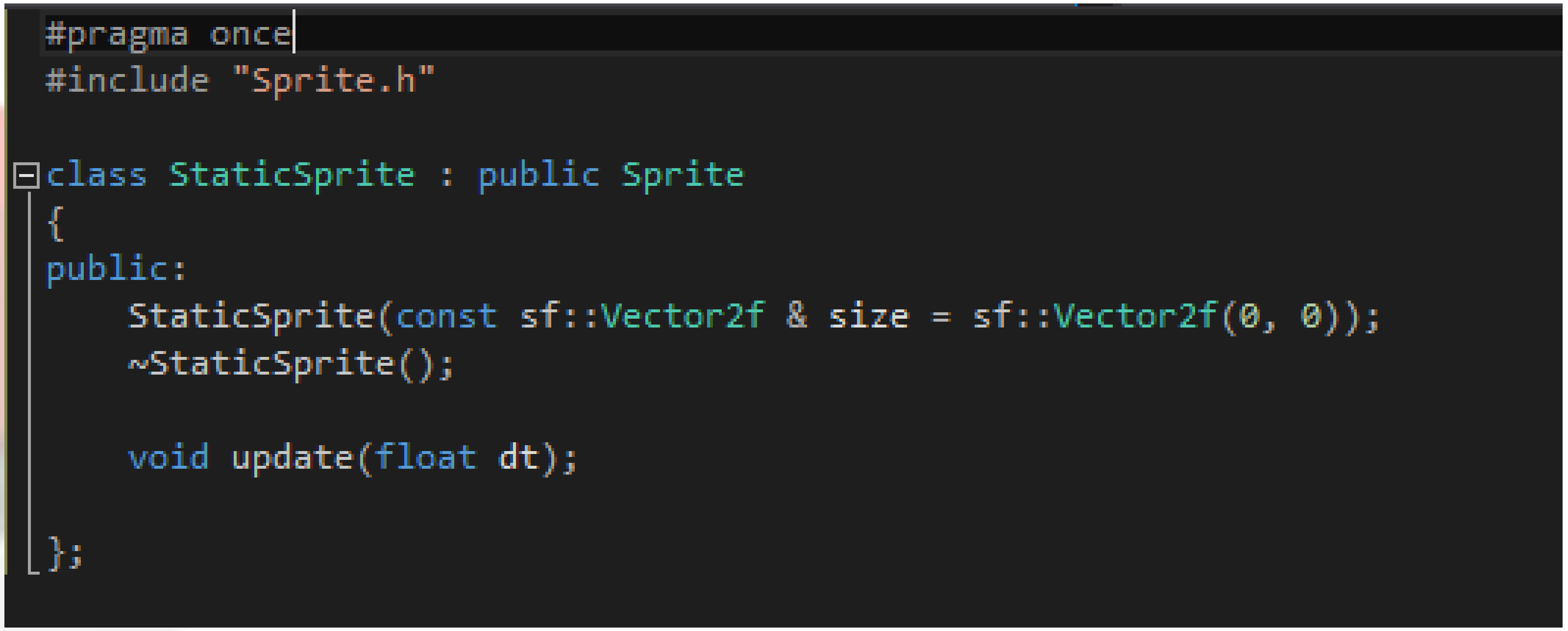
```cpp
#pragma once
#include "SFML\Graphics.hpp"

class Sprite : public sf::RectangleShape
{
public:
    Sprite(const sf::Vector2f & size = sf::Vector2f(0, 0));
    ~Sprite();

    virtual void update(float dt)=0;
    void setVelocity(sf::Vector2f vel);
    void setVelocity(float vx, float vy);
    sf::Vector2f getVelocity();

protected:
    sf::Vector2f velocity;

};
```

Using the Sprite class

- We can't use the Sprite class directly
  - Due to the pure virtual functions
- Sprite acts as a base class providing functionality for derived classes
  - And a base class for other operations
  - E.g. collision detection

Inherit from Sprite class

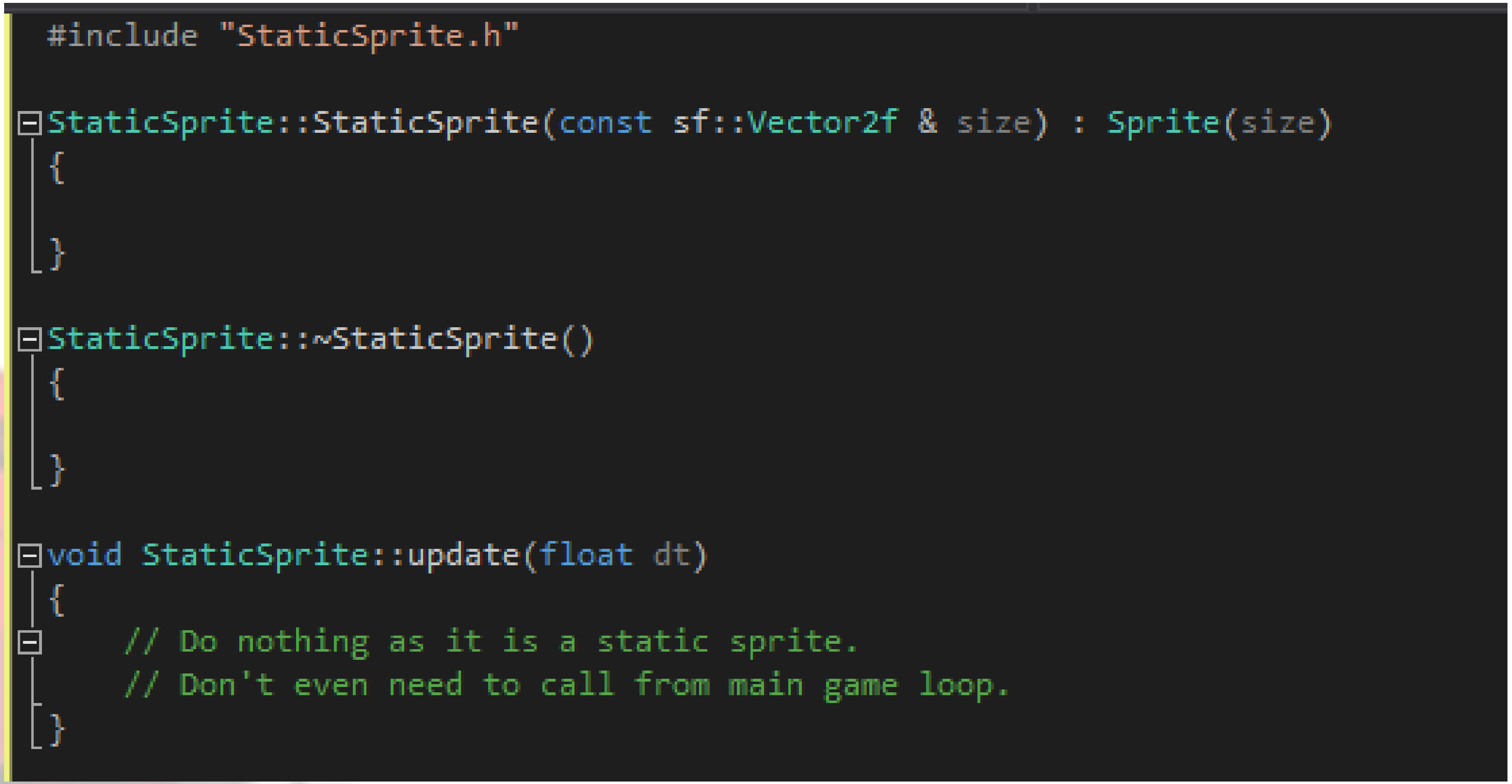
- Create a new class that inherits from Sprite
- Very simple Static Sprite class
  - Renders an image inside a rectangle
  - No additional logic or controls



```
#pragma once
#include "Sprite.h"

class StaticSprite : public Sprite
{
public:
    StaticSprite(const sf::Vector2f & size = sf::Vector2f(0, 0));
    ~StaticSprite();

    void update(float dt);

};
```

Inherit from Sprite class

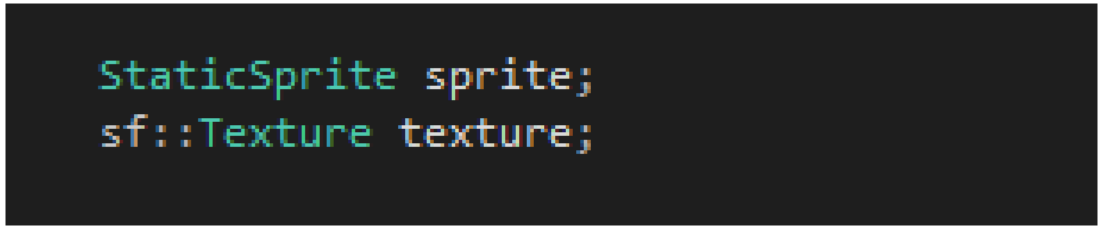
- StaticSprite.cpp



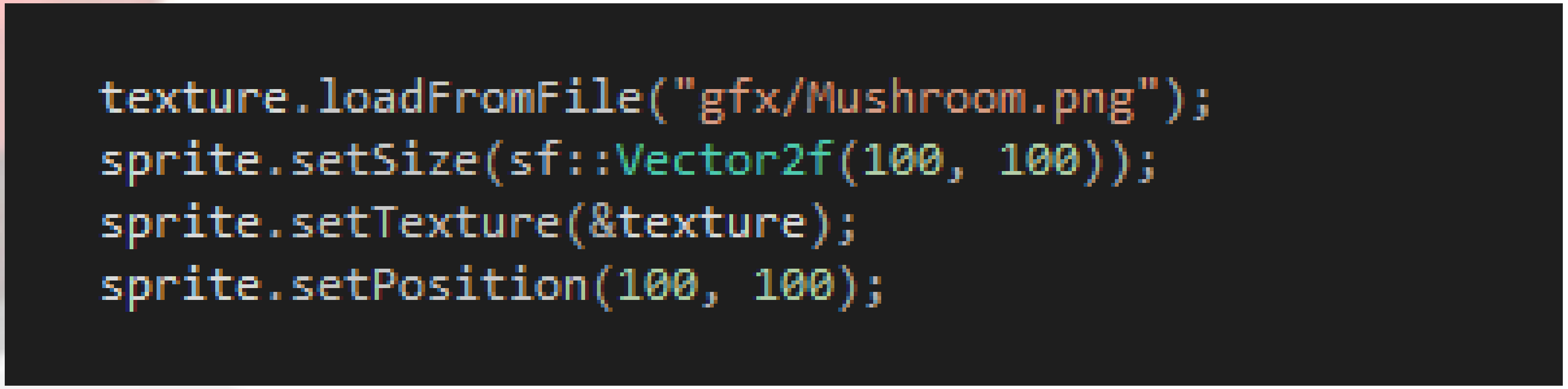
```cpp
#include "StaticSprite.h"

StaticSprite::StaticSprite(const sf::Vector2f & size) : Sprite(size)
{


}

StaticSprite::~StaticSprite()
{


}

void StaticSprite::update(float dt)
{
    // Do nothing as it is a static sprite.
    // Don't even need to call from main game loop.
}
```

Loading textures

- SFML has a texture class that will handle load images
- The supported image formats:
  - bmp, png, tga, jpg, gif, and psd.
- We will focus on working with png and jpg



```
StaticSprite sprite;
sf::Texture texture;
```

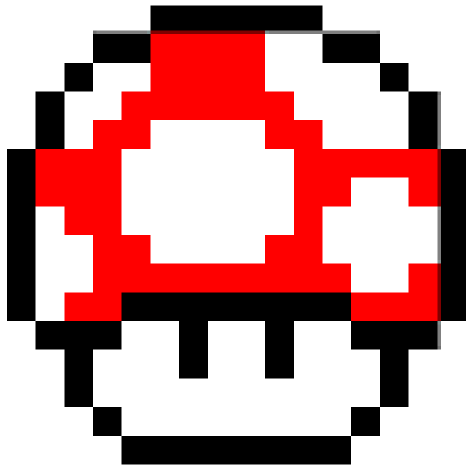


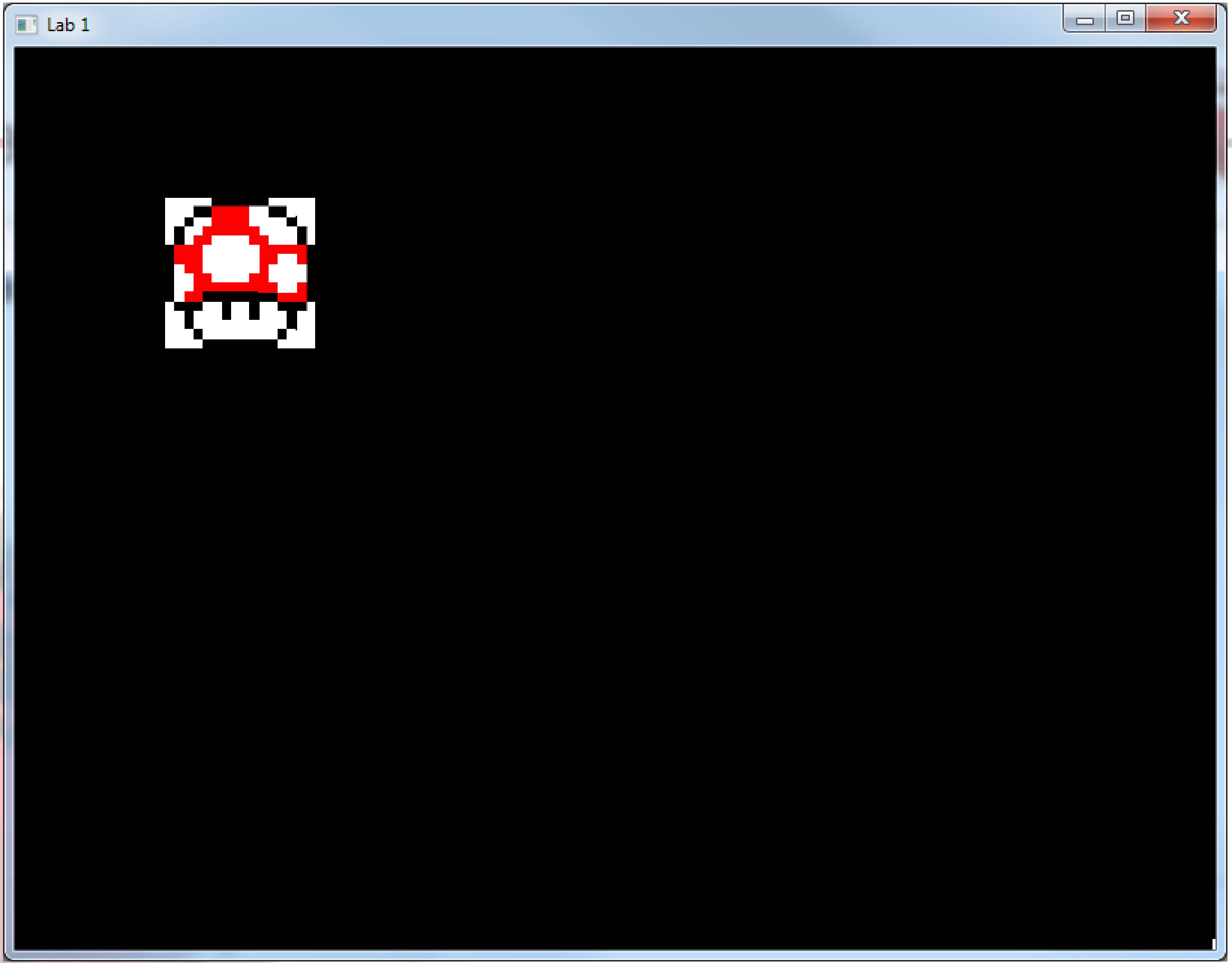
```
texture.loadFromFile("gfx/Mushroom.png");
sprite.setSize(sf::Vector2f(100, 100));
sprite.setTexture(&texture);
sprite.setPosition(100, 100);
```

Images

- Example image ->>
- 128x128 pixels

- When being rendered will shrink/stretch to fit size of object being rendered
- I will provide a few test images




Lab 1

Further reading

- SFML rectangle shape functions
  - https://www.sfml-dev.org/documentation/2.4.1/classsf_1_1RectangleShape.php
- Teach Yourself C++ in 21 Days
  - http://101.lv/learn/C++/
  - Inheritance and polymorphism
    - Day 12
    - Day 13
    - Day 14

In the labs

- Creating our sprite class
- Extending our sprite class

- Places to get sprites
  - http://www.spriters-resource.com/
  - http://opengameart.org/art-search-advanced?keys=&field_art_type_tid%5B%5D=9