Game Dev with C++: Classes, Object Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a paradigm widely used in game development due to its ability to model complex systems and interactions. In C++, classes are the cornerstone of OOP. Here's an overview of classes and OOP in C++ with a focus on game development:

Classes in C++:
A class is a blueprint for creating objects (instances) that encapsulate data and behavior. Here's a breakdown of key concepts:

1.***Data Members***: These are variables within the class that store the state of objects.

```
class Player {
private:
    int health; // Example data member
public:
    // Member functions
};
```

2.***Member Functions (Methods)***: These are functions within the class that operate on the object's data. They can manipulate the internal state and provide an interface for interacting with the object.

```
class Player {
public:
    void TakeDamage(int amount) {
        health -= amount;
    }
};
```

3.***Access Specifiers***: `private`, `public`, and `protected` determine the visibility of class members. `private` members are accessible only within the class, `public` members are accessible from outside the class, and `protected` members are accessible by derived classes.

```
class Player {
private:
    int health; // Private member
public:
    void TakeDamage(int amount) {
```

```
        health -= amount;
    } // Public member function
};
```

Object-Oriented Programming Concepts:
OOP emphasizes several principles for designing software:

1. ***Encapsulation***: Bundling data and methods that operate on the data within a single unit (class). This hides the internal workings of the class and provides a clear interface for interacting with it.

```
class Player {
private:
    int health;
public:
    Player(int initialHealth) : health(initialHealth) {}

    void TakeDamage(int amount) {
        health -= amount;
    }

    int GetHealth() const {
        return health;
    }
};
```

In this example, health is encapsulated within the Player class, and access to it is controlled by member functions (TakeDamage and GetHealth). Users of the Player class can only modify the health indirectly through the TakeDamage method, maintaining encapsulation.

2. ***Inheritance***: A mechanism that allows a class to inherit properties and behavior from another class. It promotes code reuse and allows for creating specialized classes based on existing ones.

```
class Character {
protected:
    int health;
public:
    Character(int initialHealth) : health(initialHealth) {}

    virtual void Attack() = 0; // Pure virtual function
};

class Player : public Character {
public:
    Player(int initialHealth) : Character(initialHealth) {}

    void Attack() override {
        // Player-specific attack logic
    }
};

class Enemy : public Character {
public:
    Enemy(int initialHealth) : Character(initialHealth) {}

    void Attack() override {
        // Enemy-specific attack logic
    }
};
```

In this example, both Player and Enemy classes inherit from the Character class. They share the health member and override the Attack method with their specific implementations.

3. ***Polymorphism***: The ability for objects of different classes to be treated as objects of a common superclass. This allows for more flexible and generic code.

```
class Character {
protected:
    int health;
public:
    Character(int initialHealth) : health(initialHealth) {}

    virtual void Attack() {
        // Base attack logic
    }
};

class Player : public Character {
public:
    Player(int initialHealth) : Character(initialHealth) {}

    void Attack() override {
```

```
        // Player-specific attack logic
    }
};

class Enemy : public Character {
public:
    Enemy(int initialHealth) : Character(initialHealth) {}

    void Attack() override {
        // Enemy-specific attack logic
    }
};

void DoBattle(Character* character) {
    // Perform attack
    character->Attack();
}
```

In this example, the DoBattle function takes a pointer to a Character. It can accept both Player and Enemy objects due to polymorphism. At runtime, the correct Attack method corresponding to the actual type of the object is called.

4. ***Abstraction***: The process of hiding the implementation details while showing only the essential features of the object. It helps in reducing programming complexity and effort.

```
class PhysicsEngine {
public:
    void HandleCollision(Character* object1, Character* object2) {
        // Abstracted collision handling logic
    }
};
```

In this example, the PhysicsEngine class encapsulates collision handling logic. The details of how collisions are detected and resolved are hidden from the rest of the program, providing a simpler and more abstract interface.

5. ***Composition***: The concept of building complex objects by combining simpler ones. This is achieved by including instances of other classes as members within a class.

```
class GameObject {
private:
    Transform transform;
    Renderable renderable;
    Collider collider;
public:
    void Update() {
        transform.Update();
        renderable.Render();
        collider.CheckCollisions();
```

```
    }
};
```

In this example, the GameObject class is composed of Transform, Renderable, and Collider components. Each component is responsible for a specific aspect of the game object's behavior, such as position and rendering, and collision detection, respectively. Together, they form a complete game object.

Game Programming and OOP:
In game programming, OOP is used extensively to model game entities, behaviors, and interactions. Here's how OOP concepts are applied:

1.***Entities as Objects***: Game entities like players, enemies, items, etc., are often represented as objects. Each entity can have its own properties and behaviors encapsulated within a class.

2.***Inheritance for Specialization***: Inheritance is used to create specialized entities. For example, a `Player` class may inherit from a more generic `Character` class, inheriting common properties and behaviors while adding player-specific functionality.

3.***Polymorphism for Flexibility***: Polymorphism is used to handle interactions between different types of game objects. For example, a `Render` function might take a base class pointer, allowing it to render various types of game objects without needing to know their exact type.

4.***Abstraction for Simplification***: Abstraction is used to simplify complex systems. For instance, a physics engine might abstract away the details of collision detection and resolution, providing a simple interface for game objects to interact with.

5.***Composition for Complex Objects***: Complex game objects are often built by composing simpler ones. For example, a `GameObject` class might contain components such as a `Transform`, `Renderable`, and `Collider`, each responsible for a specific aspect of the object's behavior.

By utilizing these OOP concepts, game developers can create well-structured, modular, and maintainable codebases for their games, leading to more efficient development and easier iteration.

Common g++ commands for compiling, linking, and running C++ code:

1. **Compiling C++ Code**:

   To compile a C++ source file (e.g., `example.cpp`) into an object file (e.g., `example.o`):
   ```
   g++ -c example.cpp -o example.o
   ```

   This command compiles the source file `example.cpp` into an object file named `example.o`.

2. **Linking Object Files**:

   To link one or more object files (e.g., `example.o`) into an executable (e.g., `example`):
   ```
   g++ example.o -o example
   ```

   This command links the object file(s) into an executable named `example`.

3. **Running Executable**:

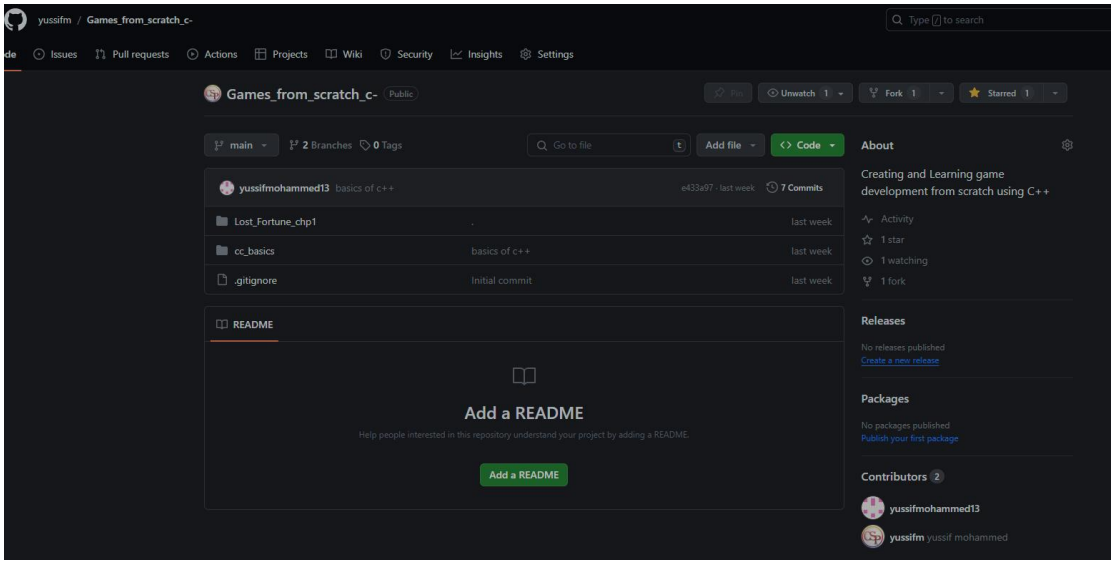
   To run the compiled executable:
   ```
   ./example
   ```

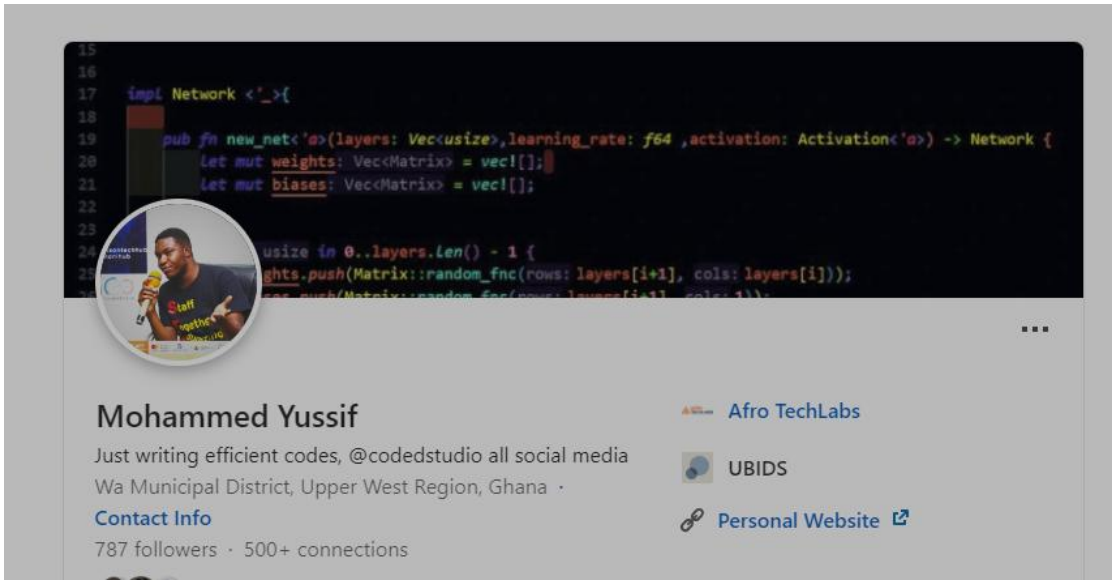
   This command executes the compiled executable named `example`.

Here's a summary:

- `**g++**`: This is the GNU Compiler Collection for compiling C++ programs.
- `**-c**`: This option tells `g++` to compile the source file(s) into object file(s) without linking.
- `**-o**`: This option specifies the output file name.
- `**./**`: This notation is used to execute the compiled executable from the current directory.

You can adjust the filenames (`example.cpp`, `example.o`, `example`) according to your actual source code file(s) and desired executable name.


yussifm / Games_from_scratch_c-
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Games_from_scratch_c-
Public
Unwatch 1
Fork 1
Starred 1
main
2 Branches
0 Tags
Go to file
Add file
Code
About
Creating and Learning game development from scratch using C++
yussifmohammed13 basics of c++
e433a97 · last week
7 Commits
Lost_Fortune_chp1
last week
cc_basics
basics of c++
last week
.gitignore
Initial commit
last week
README
Add a README
Help people interested in this repository understand your project by adding a README.
Add a README
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Contributors 2
yussifmohammed13
yussifm yussif mohammed


impl Network <'_>{
pub fn new_net<'a>(layers: Vec<usize>,learning_rate: f64 ,activation: Activation<'a>) -> Network {
let mut weights: Vec<Matrix> = vec![];
let mut biases: Vec<Matrix> = vec![];
Mohammed Yussif
Just writing efficient codes, @codedstudio all social media
Wa Municipal District, Upper West Region, Ghana ·
Contact Info
787 followers · 500+ connections
Afro TechLabs
UBIDS
Personal Website