Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice

Making your Program Performance Fly!
The Flyweight Design Pattern

-- Design Patterns Series

with Mike Shah

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

19:00 - 20:00 CES (1pm ET) Tue. March 19, 2024

60 minutes + 15 minute Q&A After Introductory Audience

Your Tour Guide for Today

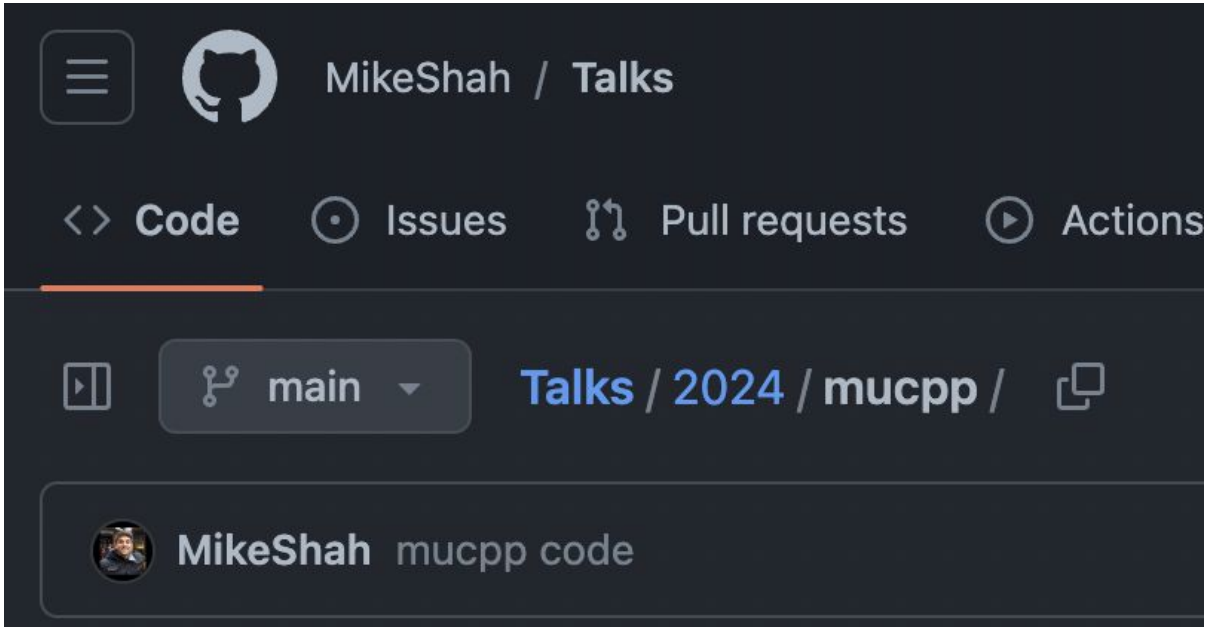
by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
  - My research is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do consulting and technical training on modern C++, DLang, Concurrency, and Graphics Programming
  - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2024/mucpp



Abstract

The abstract that you read and enticed you to join me is here!

The flyweight design pattern is a fundamental structural design pattern that allows objects to reuse or ‘cache’ shared pieces of data. One might go as far to say that the flyweight design pattern is an obvious pattern when you learn it, but I’ll share in my experience where it often only becomes obvious after building a system. In this talk I’ll introduce the flyweight, and talk about how it is used frequently in domains like computer graphics, but also useful anywhere an object is built of individual components. We’ll then discuss how to instantiate objects using the flyweight, compare flyweight objects, and the trade-offs of this pattern versus other related patterns (e.g. component systems). Attendees will leave this talk understanding how to implement the flyweight pattern and understand the trade-offs with this fundamental structural design pattern. This talk will be accessible to beginners and have Modern C++ code available.

Design Patterns

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

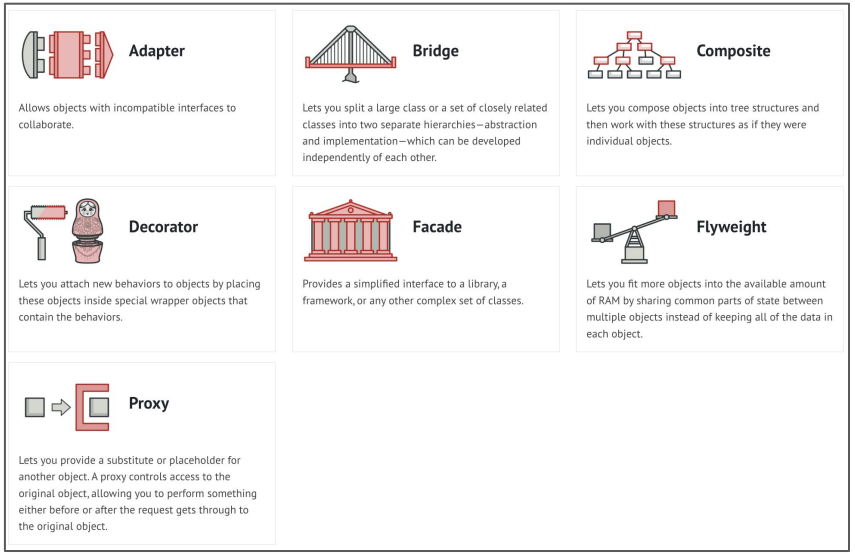
Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

https://en.wikipedia.org/wiki/Software_design_pattern

Design Patterns (1/4)

- Today we are going to be talking about a ‘design pattern’
  - Design patterns are ‘templates’ for solving a variety of common problems related to:
    - Creating objects and/or data
    - How we structure our code
    - Our how code behaves

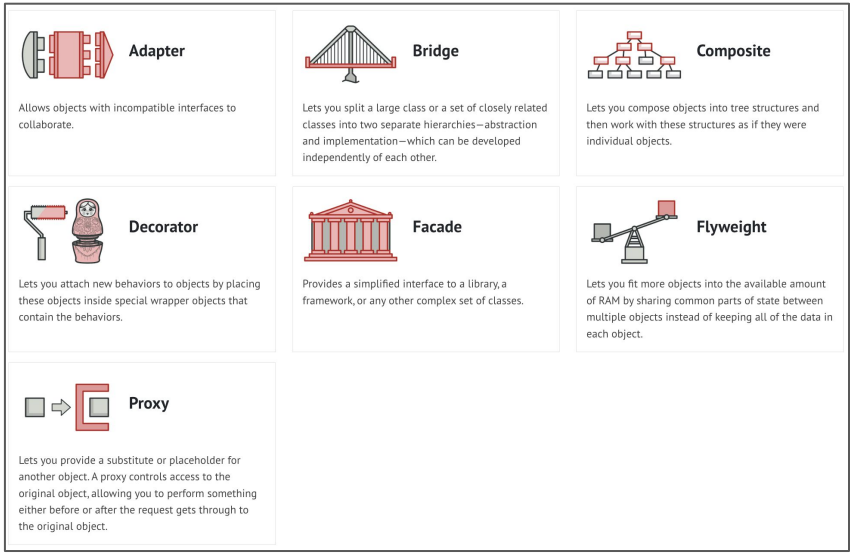


https://refactoring.guru/design-patterns/structural-patterns

Design Patterns (2/4)

- Today we are going to be talking about a ‘design pattern’
  - Design patterns are ‘templates’ for solving a variety of common problems related to:
    - Creating objects and/or data
      - Creational Design Patterns
    - How we structure our code
      - Structural Design Patterns
    - Our how code behaves
      - Behavioral Design Patterns

A popular taxonomy (i.e. organization) of design patterns is in three categories.

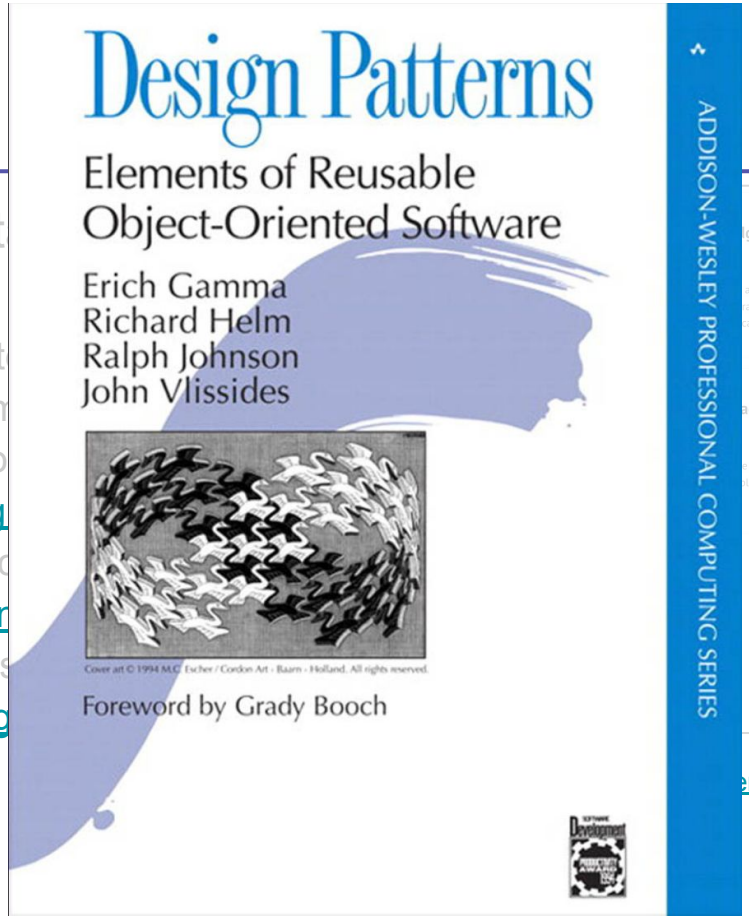


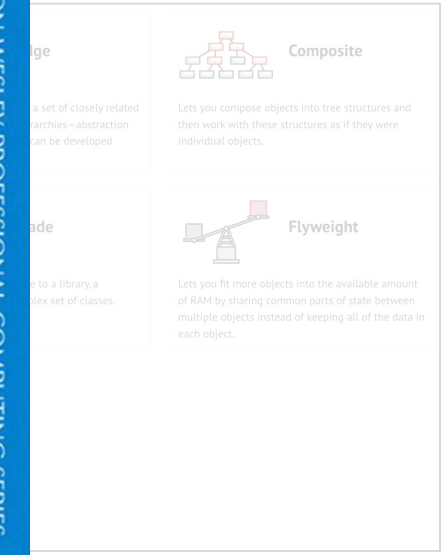
https://refactoring.guru/design-patterns/structural-patterns

Design Patterns (3/4)

- Today we are going to be t
  about a ‘design patter
  - Design patterns
    a variety of
    - g objects and/o
      Creational Desig
    - ow we structure our
      - Structural Design
    - Our how code behaves
      - Behavioral Desig

This book gets most of the credit for creating these three categories





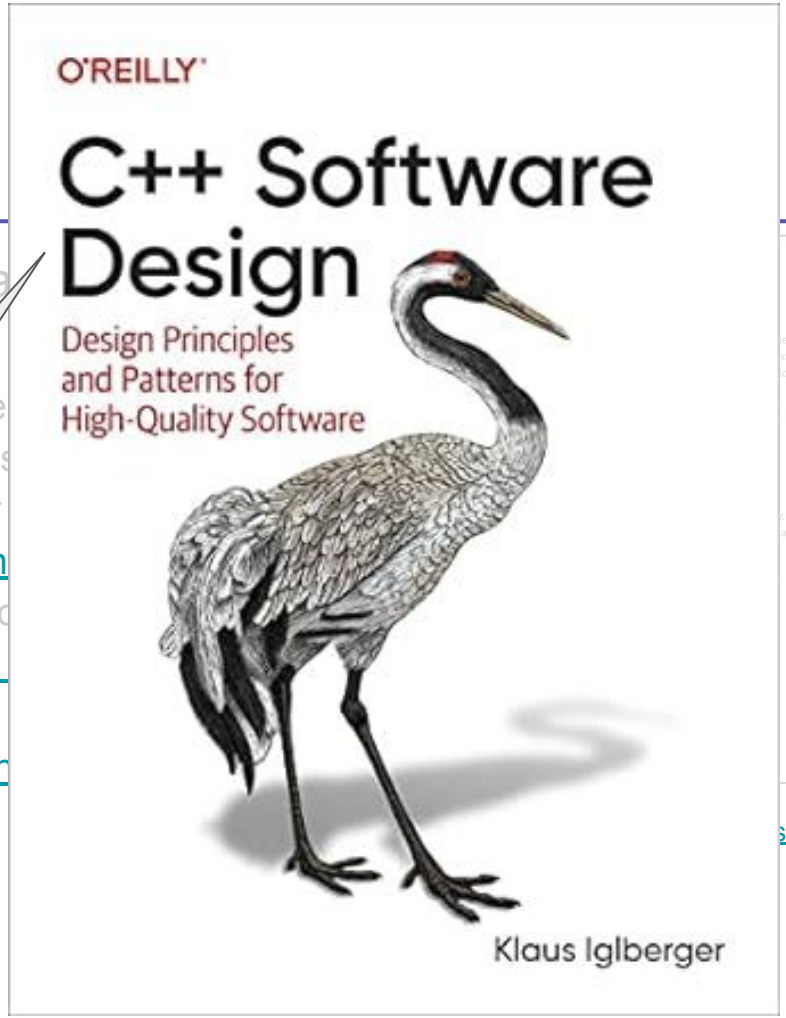
erns/structural-patterns

Design Patterns (4/4)

- Today we are going to be ta
  about a ‘design pattern’
  - Design patterns are ‘te late
    a variety of common oblems
    Creating obi s and/or
    re nal Design
    ucture our co
    uctural Design
    code behaves
    ehavioral Design

I highly recommend ‘Klaus’s’ book before/after/during looking at the Gang of Four book

He is perhaps humble -- but there are excellent samples and applied examples with Modern C++ code



structural-patterns

Recap - What is a Design Pattern (1/5)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

Recap - What is a Design Pattern (2/5)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

Recap - What is a Design Pattern (3/5)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

Recap - What is a Design Pattern 4/5)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

Recap - What is a Design Pattern (5/5)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

(Aside) Full wikipedia page -- quite a good summary! (1/3)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

(Aside) Full wikipedia page -- quite a good summary! (2/3)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transforme ow to solve a pr l best practices t ation or system.

Object-ori s or objects, w that imply mutable st e rendered unnecessa solve, and object

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

So the point of studying software design (specifically design patterns) should be to help us:

1.) Utilize a prior solution that can be shaped to help solve current problems

(Note: Perhaps as a language designer, you might also consider studying patterns to see what could be incorporated into the language!)

(Aside) Full wikipedia page -- quite a good summary! (3/3)

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a pr best practices t ation or system.

Let's take a look at a problem to better understand where a pattern may be useful!

Object-ori s or objects, w that imply mutable st rendered unnecessa solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

A Problem Domain

Game Programming

Game Complexity

- Games these days are becoming increasingly more **beautiful** and more **complex**.
  - The **beauty** in modern games is increasing in photorealism (or otherwise appealing non-photorealistic styles) due to the improvements in our hardware.
    - We have the ability to render “more” at higher resolutions
  - The **complexity** I’ve observed is from improvements in infrastructure -- i.e. the toolset (e.g. Unreal Engine) has improved our ability to focus on building data-driven games.

https://www.kotaku.com.au/wp-content/uploads/2020/08/19/kpss55oyj635psxgrrnm.gif?quality=75

Game Organization (“structure”)

- Programming games, and thinking about how to model virtual worlds is an interesting exercise.
  - An ‘object-oriented’ approach is often intuitive as it matches what we see on the screen.

Observe the individual ‘objects’ in the image to the right.

Pictured is from one of my favorite strategies games more than a decade ago -- The “Lord of the Rings: Battle for Middle-Earth”
https://i.ytimg.com/vi/wjrCvZOKyp8/maxresdefault.jpg

Game Organization (1/3)

- Let’s play a little game here
- Question to the audience:
  - What ‘attributes’ do you see of this character?
    - (i.e. what would the ‘member variables’ be if you created a ‘struct’ for this hero?)

https://www.gamespot.com/a/uploads/original/gamespot/images/2006/024/reviews/709371-929245_20060125_001.jpg

Game Organization (2/3)

- Let's play a little game here
- Question to the audience:
  - What 'attributes' do you see of this character?
    - (i.e. what would the 'member variables' be if you created a 'struct' for this hero?)
- I think something like the following would be reasonable.


```
// gameobject.cpp
struct GameObject{

    std::string name;
    Mesh        m;  // 3D data
    Texture     t;  // texture
    Position    p;  // Position
    Transform   t;  // rotation

    Behavior    b;  // Function ptr
                    // to some action

    GameObject(){ }
    ~GameObject() { }

};
```

Game Organization (3/3)

- This *may* be a perfectly reasonable implementation.
  - ‘GameObject’ serves as the ‘generic’ object that holds various attributes to create objects.
    - e.g. A ‘GameObject’ with a mesh, texture, position, and transform is a 3D character like pictured above.
    - e.g. A ‘GameObject’ with a ‘Texture’ and ‘Position’ may be a ‘dialogue’ box’
- There exist plenty of games which follow this design to build the virtual world


```
// gameobject.cpp
struct GameObject{

    std::string name;
    Mesh        m;  // 3D data
    Texture     t;  // texture
    Position    p;  // Position
    Transform   t;  // rotation

    Behavior    b;  // Function ptr
                    // to some action

    GameObject(){ }
    ~GameObject() { }

};
```

A Problem Domain

Scale in Game Programming

The Horde3D engine was used in several of these images for today's talk
https://en.wikipedia.org/wiki/Horde3D
https://github.com/horde3d/Horde3D
http://www.horde3d.org/

Managing Scale (1/8)

- As mentioned, games are growing in their beauty and their complexity
  - Let's take this example here with many 'GameObjects' (the 3D characters pictured)

http://horde3d.org/screenshots/chicago.jpg

Managing Scale (2/8)

- How many objects are instantiated here?
  - Let's say we could have 500
  - Consider those 500 objects may be the same or different
    - All those objects also carry variations of geometry, texture, mesh, names, etc. (as discussed in previous activity)

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

http://horde3d.org/screenshots/chicago.jpg

Managing Scale (3/8)

- How many objects are instantiated here
  -
  -
    in previous activity)

Keep in mind ‘500’ is even a relatively low number.

Consider each ‘blade of grass’ in this graphics demo -- where there could be thousands of objects for a relatively simple primitive.

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

https://www.gamedev.net/blogs/entry/2276570-opengl-grass-on-a-windy-day-video/

Managing Scale (4/8)

- How many objects are instantiated her
  -
  -

in previous activity)

(More neat examples)
https://www.youtube.com/watch?v=Ibe1JBF5i5Y

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

Managing Scale (5/8)

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

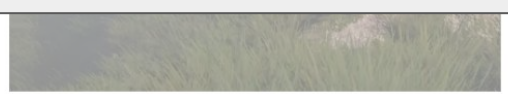
- So what is the challenge here?
  - 1. We potentially have **many copies of the same data** (on CPU and/or GPU)
  - 2. We may **want ‘unique’ attributes per object**
    - i.e. It would look weird in the simulation if all characters walked in sync
    - i.e. It would look weird if all of the grass was oriented in the same away and every blade of grass was the same size.

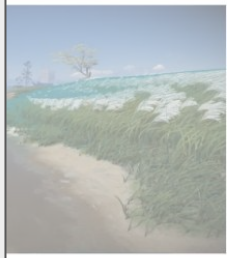
Managing Scale (6/8)

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

- So what is the challenge here?
  - 1. We potentially have **many copies of the sa**
  - 2. We **per ob**
    - i
    - i

away and every blade of grass was the same size.





Question to the audience:

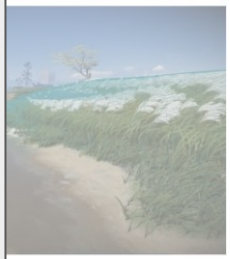
What are our programming tools to deal with these challenges?

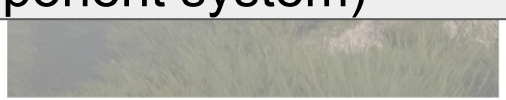
Managing Scale (7/8)

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

- So what is the challenge here?
  - 1. We potentially have **many copies of the sa**
  - 2. We **per ob**
    - i
    - i

away and every blade of grass was the same size.





Question to the audience:

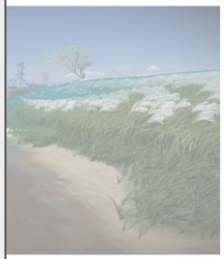
What are our programming tools to deal with these challenges?

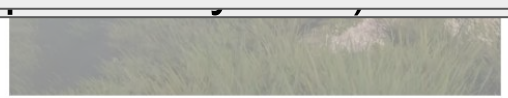
Possible answers: A mechanism for sharing (e.g. pointers, a database, component system)

Managing Scale (8/8)

```
20      // Stack may be too small or valuable
21      // to allocate our objects on
22      // GameObject[500] objects;
23
24      // Allocate on the heap?
25      GameObject* objects = new GameObject[500];
```

- So what is the challenge here?
  - 1. We potentially have **many copies of the sa**
  - 2. We **per ob**
    - i
    - i

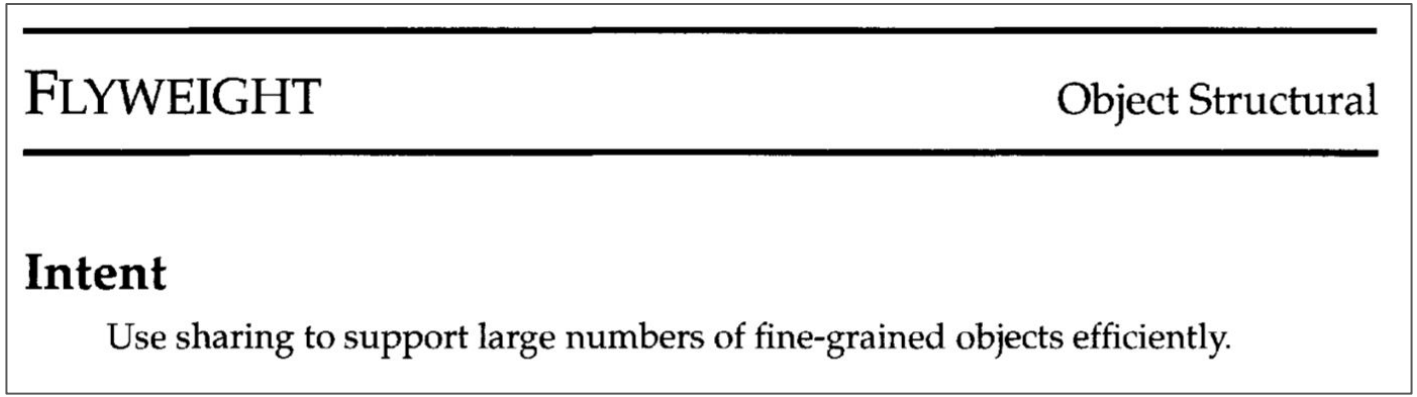




away and every blade of grass was the same size.

To help solve this issue, we have a specific pattern to help us - **The Flyweight pattern**

(and we can think about the specifics from your previous answer)



Flyweight Design Pattern

https://en.wikipedia.org/wiki/Software

(Aside) Flyweight

- Understanding the word ‘flyweight’ escapes (even as an English speaker).
  - It has some origin in the sport of boxing to mean ‘lightweight’
  - A stack overflow response speculates the term means something related to a ‘flywheel’ -- which has something to do with efficiency
- So perhaps it’s best to just use the definition given, and to understand ‘Flyweight’ is a structural design pattern

FLYWEIGHT — Object Structural

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

Professional boxing

Before 1909, anyone below featherweight was considered a bantamweight, regardless of how small the boxer. In 1911, the organization that eventually became the British Boxing Board of Control held a match that crowned Sid Smith as the first flyweight champion of the world.

Wikipedia
https://en.wikipedia.org › wiki › Flyweight

Flyweight - Wikipedia

Flywheel

A flywheel is a mechanical device which uses the conservation of angular momentum to store rotational energy; a form of kinetic energy proportional to the product of its moment of inertia and the square of its rotational speed. Wikipedia

Flyweight pattern (a structural design pattern) (1/2)

- "*A **flyweight** is a shared object that can be used in multiple contexts simultaneously* " (GOF book)
- We use the **flyweight pattern** to help us with solving our particular problem:
  - When we have a *large number of objects sharing a common properties*
    - Thus we want to save memory (i.e. space)
      - And as a side-effect, often also improve performance.

https://en.wikipedia.org/wiki/Flyweight_pattern

Flyweight pattern (a **structural design pattern**) (2/2)

- "*A **flyweight** is a shared object that can be used in multiple contexts simultaneously* " (GOF book)
- We use the **flyweight pattern** to help us with solving our particular problem:
  - When we have a *large number of objects sharing common properties*
    - Thus we want to save memory (i.e. space)
      - And as a side-effect, often also improve performance.

It will be good to understand exactly what a 'structural' design pattern is.

https://en.wikipedia.org/wiki/Flyweight_pattern

Structural Design Pattern

- In short, structural patterns are about ‘code organization’
  - Two primary mechanisms in C++ are:
    - Inheritance
    - Composition
- I’ll shortly show an example of using composition for our ‘Flyweight’

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together. Another example is the class form of the Adapter (139) pattern. In general, an adapter makes one interface (the adaptee’s) conform to another, thereby providing a uniform abstraction of different interfaces. A class adapter accomplishes this by inheriting privately from an adaptee class. The adapter then expresses its interface in terms of the adaptee’s.

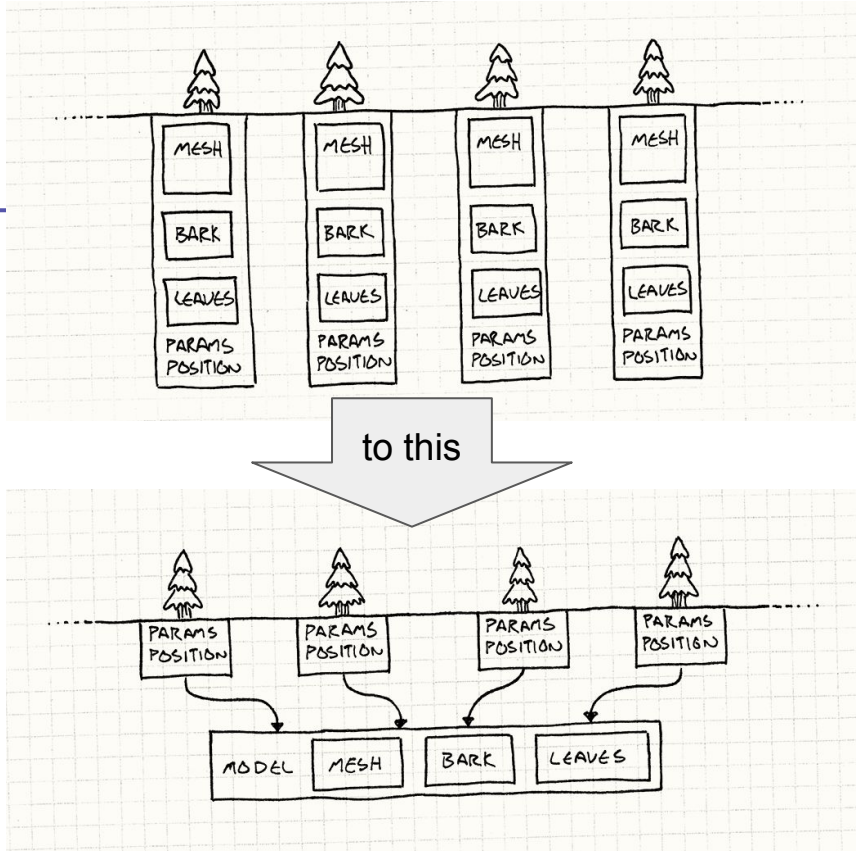
Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

P. 137 of Gang of Four Book

Example Flyweight in C++

Goal (1/2)

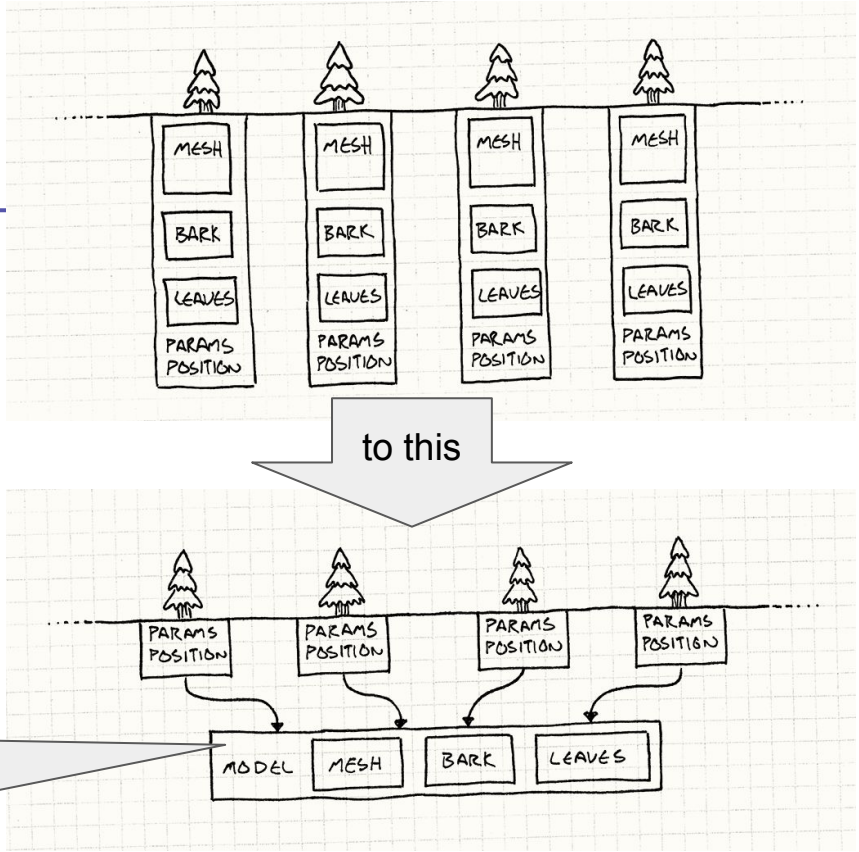
- *A **flyweight** is a shared object that can be used in multiple contexts simultaneously*
- Our Goal:
  - To create a ‘flyweight’ that can be shared amongst multiple objects



https://gameprogrammingpatterns.com/flyweight.html

Goal (2/2)

- *A **flyweight** is a shared object that can be used in multiple contexts simultaneously*
- Our Goal:
  - To create a ‘flyweight’ that can be shared amongst multiple objects

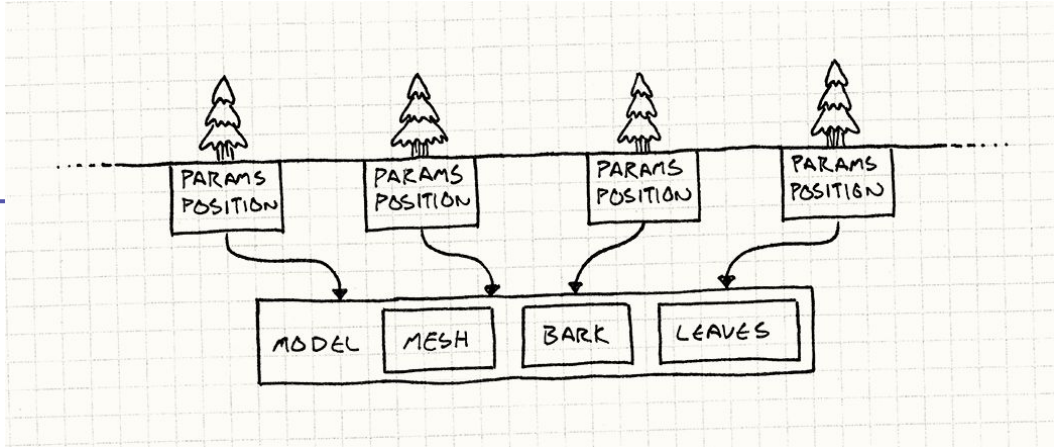


‘Model’ in this case is the ‘flyweight’ (i.e. shared) object

https://gameprogrammingpatterns.com/flyweight.html

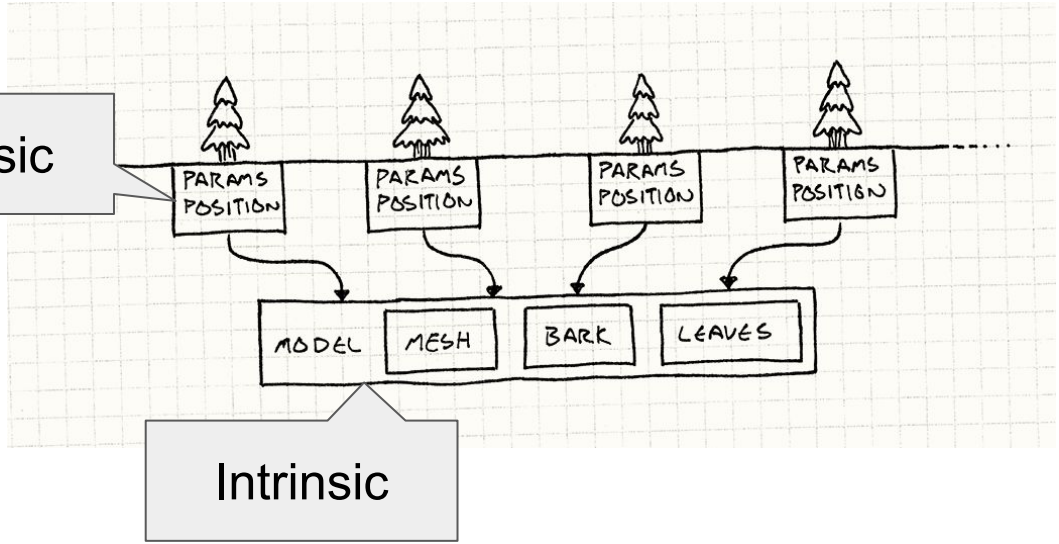
Flyweight Terms (1/2)

- In the picture:
- The **‘intrinsic’** state to the flyweight is the ‘model’
  - This could often be ‘const’ data members for instance.
  - It’s data that is not changing, thus benefits from being shared
- The **‘extrinsic’** state (unique to each tree) is the position and other params.
  - Extrinsic state can be part of the object or shared

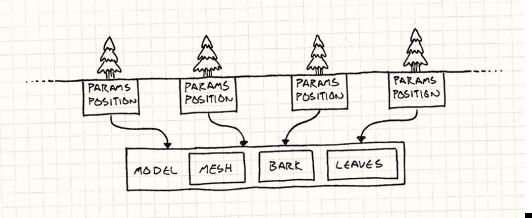


Flyweight Terms (2

- In the picture:
- The **‘intrinsic’** state to the flyweight is the ‘model’
  - This could often be ‘const’ data members for instance.
  - It’s data that is not changing, thus benefits from being shared
- The **‘extrinsic’** state (unique to each tree) is the position and other params.
  - Extrinsic state can be part of the object or shared



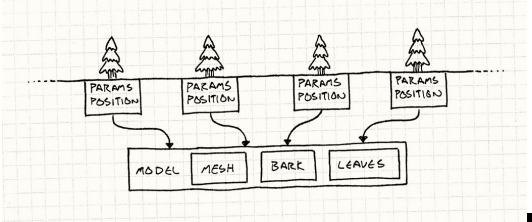
Flyweight Example (1/2)



- The following is an example of a ‘flyweight’ pattern.
  - We have now split our GameObject into two categories:
    - Extrinsic (‘often’ unique per invocation data)
    - Intrinsic (‘shared data’)
  - The goal here again is to identify with this pattern where we might be able to ‘share’ data and avoid duplication.

```cpp
// model.cpp

struct GameObject{
    // Extrinsic State
    Params params;
    Position p;
    // Intrinsic State
    //  - Shared object
    //  - likely const
    const Model* m;

};

struct Model{
    Mesh m;
    Bark b;
    Leaves l;

};
```

Flyweight Example (2/2)



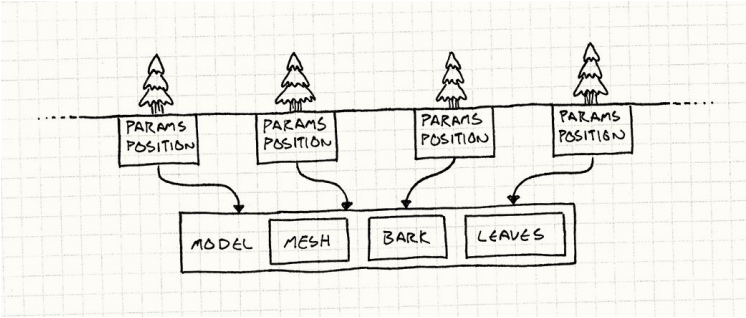
- The following is an example of a ‘flyweight’ pattern.
  - We have now split our GameObject into two categories:
    - Extrinsic (‘often’ unique per invocation data)
    - Intrinsic (‘shared data’)
  - The goal here again is to identify with this pattern where we might be able to ‘share’ data and avoid duplication.

```cpp
// model.cpp

struct GameObject{
    // Extrinsic State
    Params params;
    Position p;
    // Intrinsic State
    //   - Shared object
    //   - likely const
    const Model* m;

};

struct Model{
    Mesh m;
    Bark b;
    Leaves l;

};
```

- In my mind -- this is why this is a ‘structural’ design pattern -- which is concerned with how our objects are created.
  - (In fact -- this is sort of an ‘anti-creational’ pattern in which we’re trying to not create objects)
  - We use composition in our structure.

Flyweight Example 2

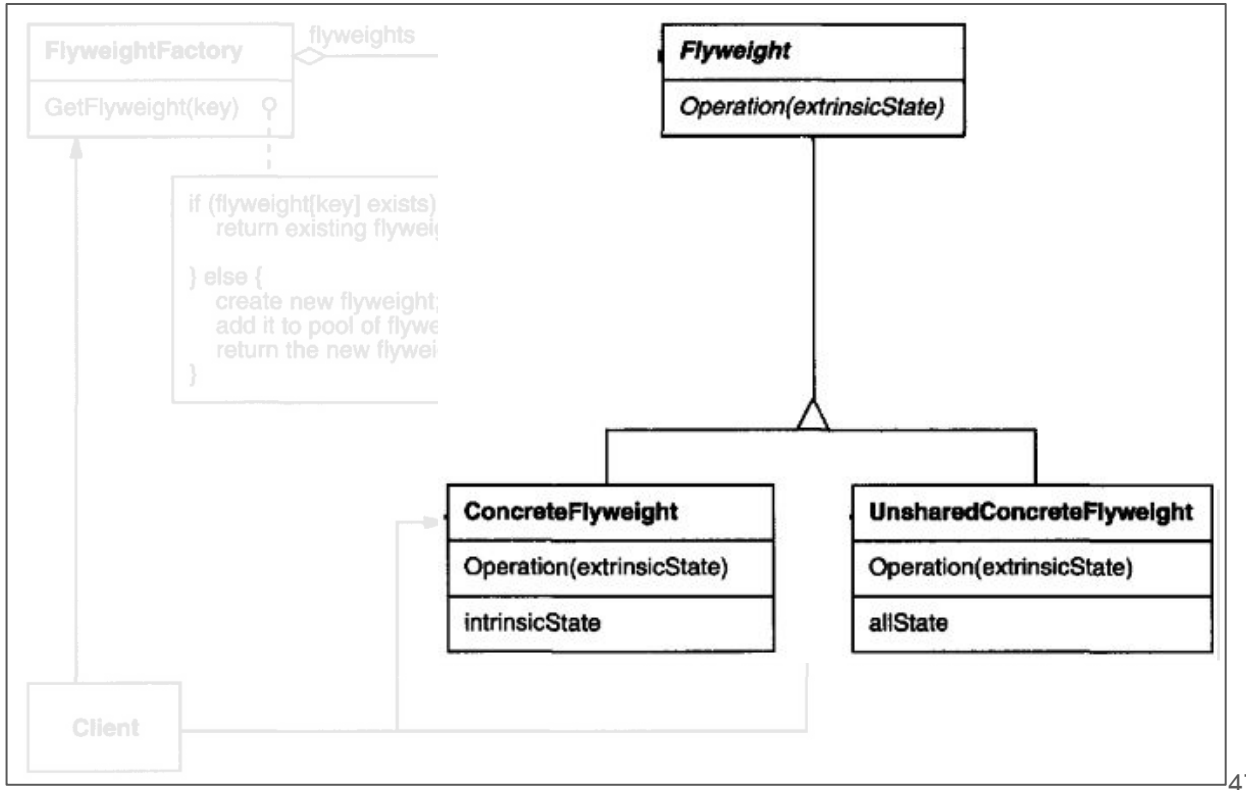
- Pictured to the right, is a sample where the flyweight (Model) holds some ‘const’ state.
  - The ‘extrinsic’ portion of the data is then passed in through a function.
  - This may help make more sense now why it is the ‘extrinsic’ (i.e. external, meant for the ‘unique’ data) state.




```
// model_extrinsic.cpp

struct ExtrinsicState{
    // Extrinsic State
    Params params;
    Position p;
};

// 'Model' is the flyweight
struct Model{
    // Inrinsic State
    const Mesh m;
    const Bark b;
    const Leaves l;

    // Extrinisc state is passed into
    // the flyweight to do the 'unique'
    // thing.
    void DrawOperation(ExtrinsicState e){
        // Draw mesh at some 'position p'
        // from the extrinisic state, but
        // the geometry itself (the mesh)
        // does not otherwise change.
    }

};
```

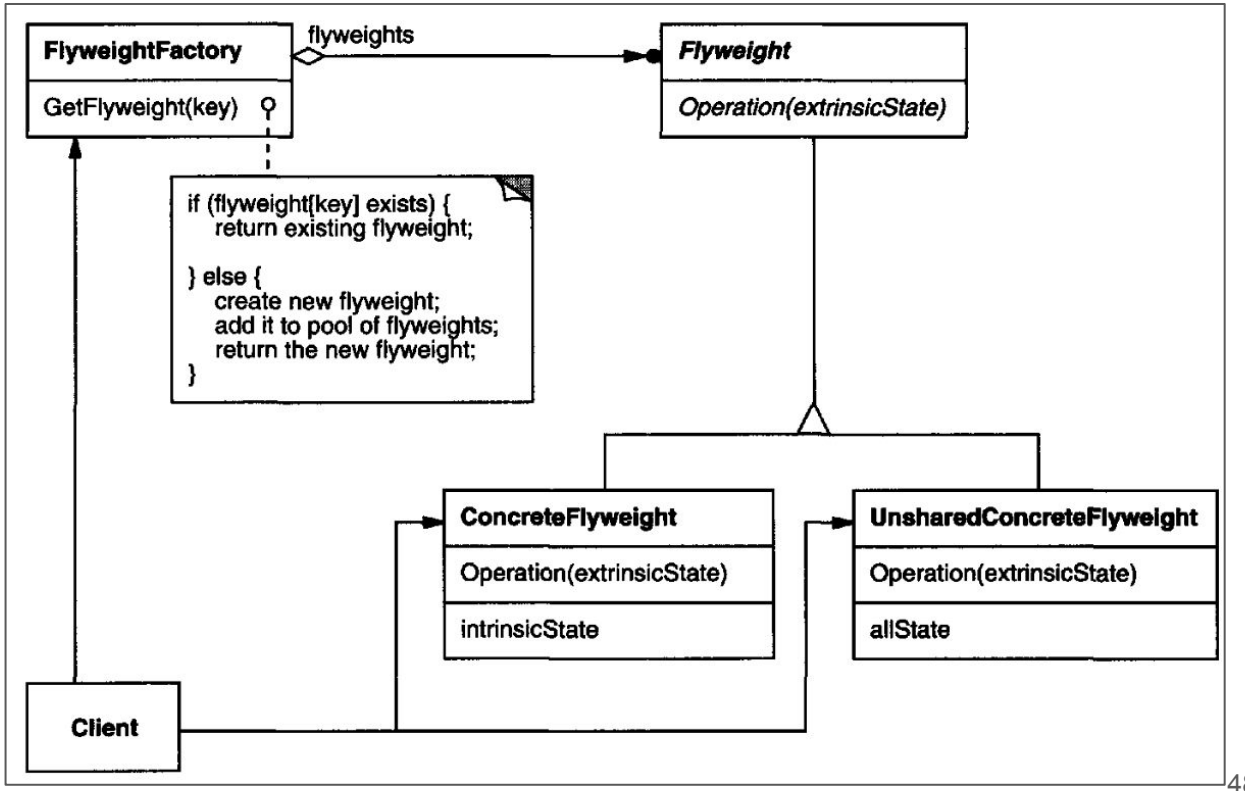
Flyweight UML Diagram

- Observe the flyweight UML diagram on the right
  - As demonstrated, we can divide our objects into intrinsic and extrinsic pieces
  - However, managing those pieces could become tricky



Flyweight UML Diagram

- The fix is to have some sort of ‘factory’ to otherwise do this.
- Note: A factory in this case could be some sort of ‘resource manager’ with a map for our flyweights.



Flyweight Factory

- Typically another way to think of the flyweight, is as a ‘resource manager’
  - i.e. We lookup objects that could be shared by some GUID (globally unique identifier), and then return that object
    - This could otherwise happen during the creation of our ‘tree’, ‘blade of grass’, etc.

```
31 // a.k.a. 'Resource Manager'
32 struct FlyWeightModelFactory{
33
34     Model* GetFlyweight(std::string key){
35         std::map<std::string,Model*>::iterator itr;
36
37         if( (itr=mFlyWeightMap.find(key)) != mFlyWeightMap.end()){
38             return itr->second;
39         }else{
40            Model* m = new Model;
41            mFlyWeightMap[key]=m;
42            return m;
43         }
44     }
45
46     // Map or another data structure to 'lookup' an exising flyweigt
47     std::map<std::string,Model*> mFlyWeightMap;
48 };
```

More Discussion on Sharing

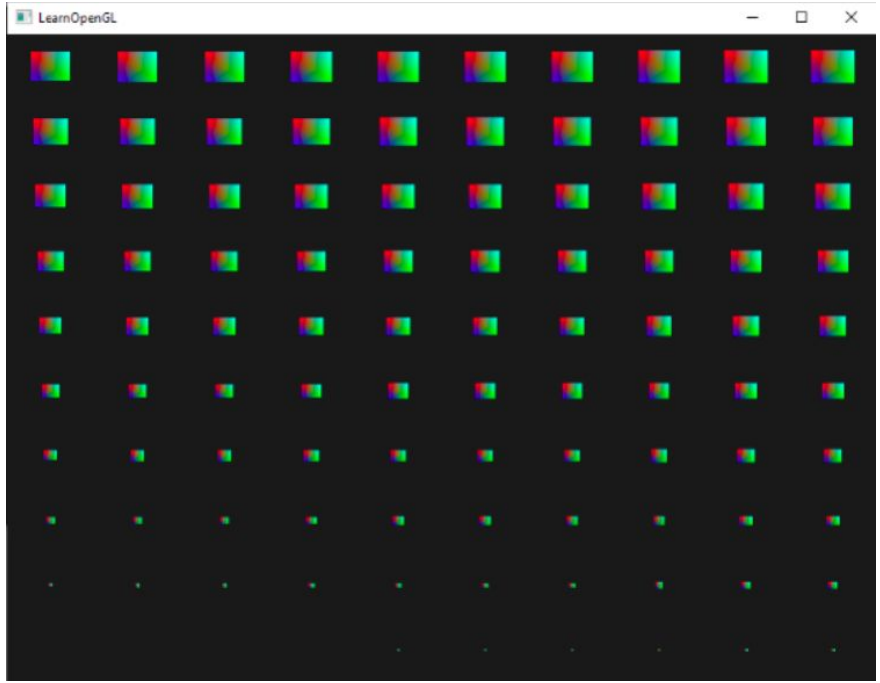
- Note:
  - We may consider other means of sharing
    - e.g. std::shared_ptr and std::weak_ptr's
- Handle System

```
31 // a.k.a. 'Resource Manager'
32 struct FlyWeightModelFactory{
33
34     Model* GetFlyweight(std::string key){
35         std::map<std::string,Model*>::iterator itr;
36
37         if( (itr=mFlyWeightMap.find(key)) != mFlyWeightMap.end()){
38             return itr->second;
39         }else{
40            Model* m = new Model;
41            mFlyWeightMap[key]=m;
42            return m;
43         }
44     }
45
46     // Map or another data structure to 'lookup' an exising flyweigt
47     std::map<std::string,Model*> mFlyWeightMap;
48 };
```

Flyweight Pattern in the Wild

Instancing in Computer Graphics

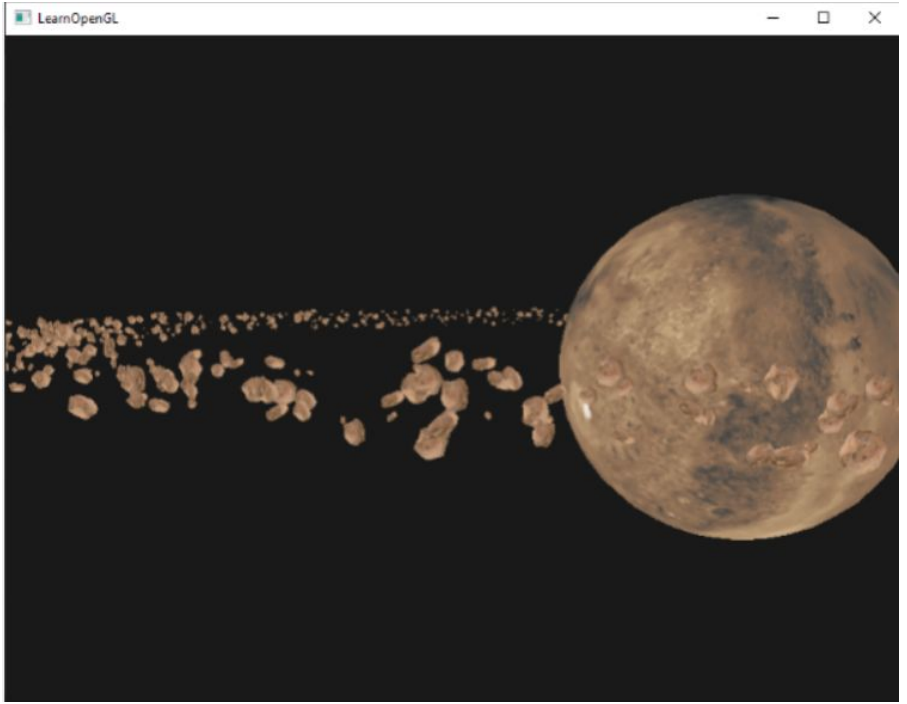
- Any time you are repeatedly using the same data, but perhaps with some variation -- that is a candidate for ‘flyweight’ pattern.
  - The case to the right is relatively obvious
    - data is the same cube
      - (Geometry and colors are the same)
    - Position and scale are the only attributes changing



https://learnopengl.com/Advanced-OpenGL/Instancing

Instancing in Computer Graphics

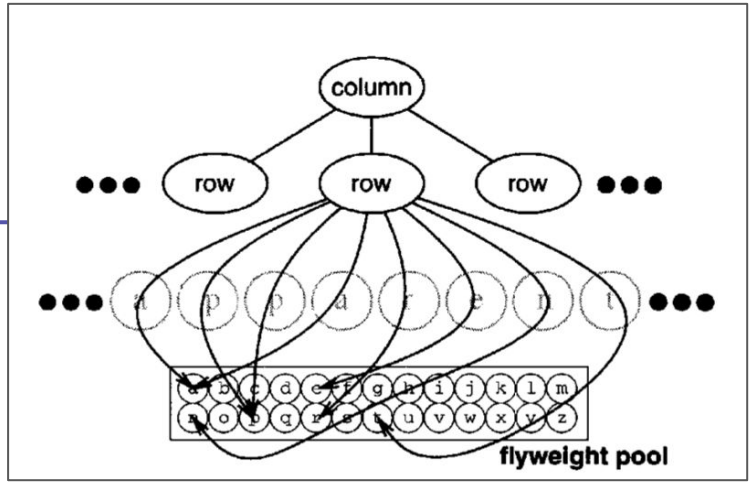
- Here's a similar example with asteroids
  - The orientation, scale, and positions are what's changing.
  - Geometry and texture however remain the same for each little piece.



https://learnopengl.com/Advanced-OpenGL/Instancing

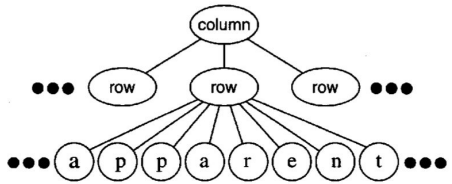
Text Rendering

- Text editing is an example used in the original Gang of Four Book
  - Each character is rendered the same way, but in different positions.



Flyweights model concepts or entities that are normally too plentiful to represent with objects. For example, a document editor can create a flyweight for each letter of the alphabet. Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is intrinsic state, while the other information is extrinsic.

Logically there is an object for every occurrence of a given character in the document:



Physically, however, there is one shared flyweight object per character, and it appears in different contexts in the document structure. Each occurrence of a particular character object refers to the same instance in the shared pool of flyweight objects:

Related Patterns

Component System

Component Pattern

- A complementary design pattern is a ‘component pattern’
  - In this system, you add attributes (as components) to an object.
  - These ‘components’ could themselves be flyweights, so as to again reduce the ‘weight’ of each individual GameObject.

```
42 /// Game Object Base Type
43 struct GameObject{
44 +--- 13 lines: / Constructor----------------------
57     void Update(){
58         // Retrieve key and value
59         for(auto& [key,value] : mComponents){
60             mComponents[key]->Update();
61         }
62     }
63 
64     template<typename T>
65     void AddComponent(T* c){
66         // Insert or update the component
67         mComponents[c->GetType()] =c;
68     }
69 
70     template<typename T>
71     T GetComponent(ComponentType type){
72         auto found = mComponents.find(type);
73         if(found != mComponents.end()){
74             return static_cast<T>(found->second);
75         }
76 
77         return nullptr;
78     }
```

```
/// GameObject Components
std::map<ComponentType,Component*> mComponents;
```

Component Example

- In the component example on the right -- what we really want from lines 14-17 is to have some ‘GetFlyweight’ to determine if the TextureComponents are unique or shared.
  - Again, we can also have some granularity as to if the components are shareable or not as well.

```
5  int main(){
6
7      // Create a Game Object
8      GameObject* obj1= new GameObject();
9      GameObject* obj2= new GameObject();
10     GameObject* obj3= new GameObject();
11     GameObject* obj4= new GameObject();
12
13     // Create components
14     TextureComponent* tex1 = new TextureComponent;
15     TextureComponent* tex2 = new TextureComponent;
16     TextureComponent* tex3 = new TextureComponent;
17     TextureComponent* tex4 = new TextureComponent;
18
19     /* More Components here */
20     // .....
21
22     // Add the component
23     obj1->AddComponent(tex1);
24     obj2->AddComponent(tex2);
25     obj3->AddComponent(tex3);
26     obj4->AddComponent(tex4);
27
28     // Push all of our objects into some data structure
29     std::vector<GameObject*> objects;
30     objects.push_back(obj1);
31     objects.push_back(obj2);
32     objects.push_back(obj3);
33     objects.push_back(obj4);
34
35     for(auto& o: objects){
36         o->Update();
37     }
38
39     return 0;
40 }
```

Related Patterns

Component System

Approximate ~~OpenGL Object Programming~~ Model (1/9)

```
// @file ooc.cpp
// g++ ooc.cpp -std=c++20 -o prog
// Object-Oriented 'C' programming
// Overly simplistic example to help you understand how OpenGL works
#include <iostream>

// Some type of OpenGL object with a unique handle and
// some data (i.e. 'array of bytes')
struct VertexBufferObject{
    int     handle;
    void*   data;
};

// Global OpenGL context
// Keeps track of all 'state' and 'objects'
struct glContext{
    VertexBufferObject VBO[100];// Assume we can only have 100 objects
    int currentFreeHandle    =0;    // Keeps track of next free identifier
                                    // and increments by 1 every time we
                                    // add a new object.
};

// Allocate a global for the context
glContext globalContext;
```

```

/// handle is like a unique 'id'
int GenerateVertexBufferObject(int* handle,void* data){
    // OpenGL context automatically finds a free 'id' or 'handle'
    // to allocate a new object.
    int current = globalContext.currentFreeHandle;
    *handle = current;

    // Assign this unique handle to our object.
    globalContext.VBO[current].handle = current;
    globalContext.VBO[current].data   = data;

    // Increment
    globalContext.currentFreeHandle++;

    return 0; // Return 0 for success, or some error code.
}

int useVBO(int handle){
    globalContext.VBO[handle].data; // Do something with data
    std::cout << "Doing something with handle..." << std::endl;

    return 0;
}

int main(){
    int someHandle1;
    int someHandle2;
    int someHandle3;
    GenerateVertexBufferObject(&someHandle1, NULL);
    GenerateVertexBufferObject(&someHandle2, NULL);
    GenerateVertexBufferObject(&someHandle3, NULL);

    std::cout << "someHandle3 = " << someHandle3 << std::endl;
    return 0;
}
```

*A
O
M

- In this example I'll show *roughly* how objects work in OpenGL.

```
// @file ooc.cpp
// g++ ooc.cpp -std=c++20 -o prog
// Object-Oriented 'C' programming
// Overly simplistic example to help you understand how OpenGL works
#include <iostream>

// Some type of OpenGL object with a unique handle and
// some data (i.e. 'array of bytes')
struct VertexBufferObject{
    int     handle;
    void*   data;
};

// Global OpenGL context
// Keeps track of all 'state' and 'objects'
struct glContext{
    VertexBufferObject VBO[100];// Assume we can only have 100 objects
    int currentFreeHandle    =0;    // Keeps track of next free identifier
                                    // and increments by 1 every time we
                                    // add a new object.
};

// Allocate a global for the context
glContext globalContext;
```

```

/// handle is like a unique 'id'
int GenerateVertexBufferObject(int* handle,void* data){
    // OpenGL context automatically finds a free 'id' or 'handle'
    // to allocate a new object.
    int current = globalContext.currentFreeHandle;
    *handle = current;

    // Assign this unique handle to our object.
    globalContext.VBO[current].handle = current;
    globalContext.VBO[current].data   = data;

    // Increment
    globalContext.currentFreeHandle++;

    return 0; // Return 0 for success, or some error code.
}

int useVBO(int handle){
    globalContext.VBO[handle].data; // Do something with data
    std::cout << "Doing something with handle..." << std::endl;

    return 0;
}

int main(){
    int someHandle1;
    int someHandle2;
    int someHandle3;
    GenerateVertexBufferObject(&someHandle1, NULL);
    GenerateVertexBufferObject(&someHandle2, NULL);
    GenerateVertexBufferObject(&someHandle3, NULL);

    std::cout << "someHandle3 = " << someHandle3 << std::endl;
    return 0;
}
```

- In OpenGL there is the ‘OpenGL context’ which is effectively a global structure keeping track of all state.
  - OpenGL itself is a giant state machine.

```
1  // @fil
2  // g++                    +20 -o prog
3  // Obje                   C' programming
4  // Over                   c example to help you understand how OpenGL works
5  #includ                   >
6
7  // Some                   OpenGL object with a unique handle and
8  // some                   (i.e. 'array of bytes')
9  struct                    xBufferObject{
10     int                    handle;
11     voi                    data;
12 };
13
14 // Global OpenGL context
15 // Keeps track of all 'state' and 'objects'
16 struct glContext{
17     VertexBufferObject VBO[100];// Assume we can only have 100 objects
18     int currentFreeHandle    =0;    // Keeps track of next free identifier
19                                     // and increments by 1 every time we
20                                     // add a new object.
21 };
22
23 // Allocate a global for the context
24 glContext globalContext;
```

```
25
26 /// handle is like a unique 'id'
27 int GenerateVertexBufferObject(int* handle,void* data){
28     // OpenGL context automatically finds a free 'id' or 'handle'
29     // to allocate a new object.
30     int current = globalContext.currentFreeHandle;
31     *handle = current;
32
33     // Assign this unique handle to our object.
34     globalContext.VBO[current].handle = current;
35     globalContext.VBO[current].data   = data;
36
37     // Increment
38     globalContext.currentFreeHandle++;
39
40     return 0; // Return 0 for success, or some error code.
41 }
42
43 int useVBO(int handle){
44     globalContext.VBO[handle].data; // Do something with data
45     std::cout << "Doing something with handle..." << std::endl;
46
47     return 0;
48 }
49
50 int main(){
51     int someHandle1;
52     int someHandle2;
53     int someHandle3;
54     GenerateVertexBufferObject(&someHandle1, NULL);
55     GenerateVertexBufferObject(&someHandle2, NULL);
56     GenerateVertexBufferObject(&someHandle3, NULL);
57
58     std::cout << "someHandle3 = " << someHandle3 << std::endl;
59     return 0;
60 }
```

- ObjectGL ‘Objects’ may hold 3D data, pixels, shaders or other information
  - Based off this name -- observe that we are storing ‘vertex information’
  - Data is usually stored in ‘flat buffers’ (i.e. 1D-arrays)

```
1  // @file ooc.c
2  // g++ ooc.cpp -o prog
3  // Object-Oriented programming
4  // Overly simplistic example to help you understand how OpenGL works
5  #include <iostream>
6
7  // Some type of OpenGL object with a unique handle and
8  // some data (i.e. 'array of bytes')
9  struct VertexBufferObject{
10     int     handle;
11     void*   data;
12 };
13
14 // Global OpenGL context
15 // Keeps track of all 'state' and 'objects'
16 struct glContext{
17     VertexBufferObject VBO[100];// Assume we can only have 100 objects
18     int currentFreeHandle    =0;    // Keeps track of next free identifier
19                                 // and increments by 1 every time we
20                                 // add a new object.
21 };
22
23 // Allocate a global for the context
24 glContext globalContext;
```

```
25
26 /// handle is like a unique 'id'
27 int GenerateVertexBufferObject(int* handle,void* data){
28     // OpenGL context automatically finds a free 'id' or 'handle'
29     // to allocate a new object.
30     int current = globalContext.currentFreeHandle;
31     *handle = current;
32
33     // Assign this unique handle to our object.
34     globalContext.VBO[current].handle = current;
35     globalContext.VBO[current].data   = data;
36
37     // Increment
38     globalContext.currentFreeHandle++;
39
40     return 0; // Return 0 for success, or some error code.
41 }
42
43 int useVBO(int handle){
44     globalContext.VBO[handle].data; // Do something with data
45     std::cout << "Doing something with handle..." << std::endl;
46
47     return 0;
48 }
49
50 int main(){
51     int someHandle1;
52     int someHandle2;
53     int someHandle3;
54     GenerateVertexBufferObject(&someHandle1, NULL);
55     GenerateVertexBufferObject(&someHandle2, NULL);
56     GenerateVertexBufferObject(&someHandle3, NULL);
57
58     std::cout << "someHandle3 = " << someHandle3 << std::endl;
59     return 0;
60 }
```

- ‘OpenGL Objects’ are identified by a ‘handle’ (i.e. integer) into an array of the global glContext object.
  - The handle corresponds to the index in the array in the OpenGL context
  - (Note: OpenGL likely does something more intelligent than using a fixed-size array of 100 VertexBufferObject’s -- this is just a demo!)

```
1  // @file ooc.c
2  // g++ ooc.cpp                    prog
3  // Object-Ori                     ogramming
4  // Overly sim                     example to help you understand how OpenGL works
5  #include <io                      m>
6
7  // Some type of OpenGL object with a unique handle and
8  // some data (i.e. 'array of bytes')
9  struct VertexBufferObject{
10     int     handle;
11     void*   data;
12 };
13
14 // Global OpenGL context
15 // Keeps track of all 'state' and 'objects'
16 struct glContext{
17     VertexBufferObject VBO[100]; // Assume we can only have 100 objects
18     int currentFreeHandle    =0;    // Keeps track of next free identifier
19                                     // and increments by 1 every time we
20                                     // add a new object.
21 };
22
23 // Allocate a global for the context
24 glContext globalContext;
```

```
                                                    oid* data){
                                                    free 'id' or 'handle'

                                                    andle;


                                                    ct.
                                                    ent;
35     globalContext.VBO[current].data   = data;
36
37     // Increment
38     globalContext.currentFreeHandle++;
39
40     return 0; // Return 0 for success, or some error code.
41 }
42
43 int useVBO(int handle){
44     globalContext.VBO[handle].data; // Do something with data
45     std::cout << "Doing something with handle..." << std::endl;
46
47     return 0;
48 }
49
50 int main(){
51     int someHandle1;
52     int someHandle2;
53     int someHandle3;
54     GenerateVertexBufferObject(&someHandle1, NULL);
55     GenerateVertexBufferObject(&someHandle2, NULL);
56     GenerateVertexBufferObject(&someHandle3, NULL);
57
58     std::cout << "someHandle3 = " << someHandle3 << std::endl;
59     return 0;
60 }
```

- You then ‘use’ the handles in functions which will access the correct OpenGL object at an array index that has been previously allocated

```cpp
// @file ooc.cpp
// g++ ooc.cpp -std=c++20 -o prog
// Object-Oriented 'C' programming
// Overly simplistic example to help you understand how OpenGL works
#include <iostream>

// Some type of OpenGL object with a unique handle and
// some data (i.e. 'array of bytes')
struct VertexBufferObject{
    int     handle;
    void*   data;
};

// Global OpenGL context
// Keeps track of all 'state' and 'objects'
struct glContext{
    VertexBufferObject VBO[100]; // Assume we can only have 100 objects
    int currentFreeHandle    =0;    // Keeps track of next free identifier
                                // and increments by 1 every time we
                                // add a new object.
};

// Allocate a global for the context
glContext globalContext;
```

```cpp
/// handle is like a unique 'id'
int GenerateVertexBufferObject(int* handle, void* data){
    // OpenGL context automatically finds a free 'id' or 'handle'
    // to allocate a new object.
    int current = globalContext.currentFreeHandle;
    *handle = current;

    // Assign this unique handle to our object.
    globalContext.VBO[current].handle = current;
    globalContext.VBO[current].data   = data;

    // Increment
    globalContext.currentFreeHandle++;

    return 0; // Return 0 for success, or some error code.
}

int useVBO(int handle)
    globalContext.VBO[handle].data; // Do something with data
    std::cout << "Doing something with handle..." << std::endl;

    return 0;
}

int main(){
    int someHandle1;
    int someHandle2;
    int someHandle3;
    GenerateVertexBufferObject(&someHandle1, NULL);
    GenerateVertexBufferObject(&someHandle2, NULL);
    GenerateVertexBufferObject(&someHandle3, NULL);

    std::cout << "someHandle3 = " << someHandle3 << std::endl;
    return 0;
}
```

- Note: A common pattern you'll see in OpenGL for functions of the form 'Gen' (short for generate) or 'Create' will be to take a pointer to an integer handle.
  - Observe that at line '51' we create an integer with no assigned value
  - At line '54' we pass in the address of 'someHandle1' into the function.
    - Within 'GenerateVertexBufferObject' the value 'someHandle1' will then be assigned through the pointer (line 31)
- You need to actually watch this entire video you do not understand -- [Learn and understand (almost) everything about the fundamentals of C++ pointers in 96 minutes]

```
22
23 // Allocate a global for the context
24 glContext globalContext;
```

```
25
26 /// handle is like a unique 'id'
27 int GenerateVertexBufferObject(int* handle, void* data){
28     // OpenGL context automatically finds a free 'id' or 'handle'
29     // to allocate a new object.
30     int current = globalContext.currentFreeHandle;
31     *handle = current;
32
33     // Assign this unique handle to our object.
34     globalContext.VBO[current].handle = current;
35     globalContext.VBO[current].data   = data;
36
37     // Increment
38     globalContext.currentFreeHandle++;
39
40     return 0; // Return 0 for success, or some error code.
41 }
42
43 int useVBO(int handle){
44     globalContext.VBO[handle].data; // Do something with data
45     std::cout << "Doing something with handle..." << std::endl;
46
47     return 0;
48 }
49
50 int main(){
51     int someHandle1;
52     int someHandle2;
53     int someHandle3;
54     GenerateVertexBufferObject(&someHandle1, NULL);
55     GenerateVertexBufferObject(&someHandle2, NULL);
56     GenerateVertexBufferObject(&someHandle3, NULL);
57
58     std::cout << "someHandle3 = " << someHandle3 << std::endl;
59     return 0;
60 }
```

- Passing in a ‘handle’ (or sometimes also called ‘name’ in OpenGL functions -- but still usually some int type) results in accessing memory from our allocated buffer in the OpenGL context.
  - Note: In most versions of OpenGL, per object type, we only have ‘1’ object type ‘bound’ at a given time
    - All proceeding operations act on the currently bound object.

```
// Some type of OpenGL object with a unique handle and
// some data (i.e. 'array of bytes')
struct VertexBufferObject{
    int     handle;
    void*   data;
};

// Global OpenGL context
// Keeps track of all 'state' and 'objects'
struct glContext{
    VertexBufferObject VBO[100];// Assume we can only have 100 objects
    int currentFreeHandle    =0;    // Keeps track of next free identifier
                                    // and increments by 1 every time we
                                    // add a new object.
};

// Allocate a global for the context
glContext globalContext;
```

```
/// handle is like a unique 'id'
int GenerateVertexBufferObject(int* handle,void* data){
    // OpenGL context automatically finds a free 'id' or 'handle'
    // to allocate a new object.
    int current = globalContext.currentFreeHandle;
    *handle = current;

    // Assign this unique handle to our object.
    globalContext.VBO[current].handle = current;
    globalContext.VBO[current].data   = data;

    // Increment
    globalContext.currentFreeHandle++;

    return 0; // Return 0 for success, or some error code.
}

int useVBO(int handle){
    globalContext.VBO[handle].data; // Do something with data
    std::cout << "Doing something with handle..." << std::endl;

    return 0;
}

int main(){
    int someHandle1;
    int someHandle2;
    int someHandle3;
    GenerateVertexBufferObject(&someHandle1, NULL);
    GenerateVertexBufferObject(&someHandle2, NULL);
    GenerateVertexBufferObject(&someHandle3, NULL);

    std::cout << "someHandle3 = " << someHandle3 << std::endl;
    return 0;
}
```

1. Again -- Often OpenGL is managing these buffers smarter for performance
2. Yes -- you could write your own map data structure to map integers to strings if you want more descriptive names.

```cpp
// @file ooc.cpp
// g++ ooc.cpp -std=c++20 -o prog
// Object-Oriented 'C' programming
// Overly simplistic example to help you understand how OpenGL works
#include <iostream>

// Some type of OpenGL object with a unique handle and
// some data (i.e. 'array of bytes')
struct VertexBufferObject{
    int     handle;
    void*   data;
};

// Global OpenGL context
// Keeps track of all 'state' and 'objects'
struct glContext{
    VertexBufferObject VBO[100];// Assume we can only have 100 objects
    int currentFreeHandle    =0;    // Keeps track of next free identifier
                                    // and increments by 1 every time we
                                    // add a new object.
};

// Allocate a global for the context
glContext globalContext;
```

```cpp

/// handle is like a unique 'id'
int GenerateVertexBufferObject(int* handle,void* data){
    // OpenGL context automatically finds a free 'id' or 'handle'
    // to allocate a new object.
    int current = globalContext.currentFreeHandle;
    *handle = current;

    // Assign this unique handle to our object.
    globalContext.VBO[current].handle = current;
    globalContext.VBO[current].data   = data;

    // Increment
    globalContext.currentFreeHandle++;

    return 0; // Return 0 for success, or some error code.
}

int useVBO(int handle){
    globalContext.VBO[handle].data; // Do something with data
    std::cout << "Doing something with handle..." << std::endl;

    return 0;
}

int main(){
    int someHandle1;
    int someHandle2;
    int someHandle3;
    GenerateVertexBufferObject(&someHandle1, NULL);
    GenerateVertexBufferObject(&someHandle2, NULL);
    GenerateVertexBufferObject(&someHandle3, NULL);

    std::cout << "someHandle3 = " << someHandle3 << std::endl;
    return 0;
}
```

Handle System

- So what I have shown you -- this idea with a ‘handle’ to an object is exactly making use of flyweights.
  - OpenGL (and other frameworks) that are ‘state machines’ may also do very well with this idea of sharing data.

```
25
26 /// handle is like a unique 'id'
27 int GenerateVertexBufferObject(int* handle,void* data){
28     // OpenGL context automatically finds a free 'id' or 'handle'
29     // to allocate a new object.
30     int current = globalContext.currentFreeHandle;
31     *handle = current;
32
33     // Assign this unique handle to our object.
34     globalContext.VBO[current].handle = current;
35     globalContext.VBO[current].data   = data;
36
37     // Increment
38     globalContext.currentFreeHandle++;
39
40     return 0; // Return 0 for success, or some error code.
41 }
42
43 nt useVBO(int handle){
44     globalContext.VBO[handle].data; // Do something with data
45     std::cout << "Doing something with handle..." << std::endl;
46
47     return 0;
48 }
49
50 int main(){
51     int someHandle1;
52     int someHandle2;
53     int someHandle3;
54     GenerateVertexBufferObject(&someHandle1, NULL);
55     GenerateVertexBufferObject(&someHandle2, NULL);
56     GenerateVertexBufferObject(&someHandle3, NULL);
57
58     std::cout << "someHandle3 = " << someHandle3 << std::endl;
59     return 0;
60 }
```

Summary

Pros and Cons

- Pros
  - Can greatly increase the performance of your program
    - Both in terms of memory usage being reduced and actual performance of application (shared data provides potentially good temporal locality)
- Neutral
  - Because we are ‘sharing’ resources ‘consistency’ is a byproduct, which may generate a ‘more correct’ result (i.e. All of our geometry is the same in a 3D mesh)
- Cons
  - You loose fine grain control of every single object
  - Some additional complexity added
    - (e.g. Resource managers/factories and the division of objects into intrinsic and extrinsic state)

More Resources

- https://gameprogrammingpatterns.com/flyweight.html
- https://www.boost.org/doc/libs/1_84_0/libs/flyweight/doc/tutorial/index.html

Thank you  2024!

Making your Program Performance Fly!
The Flyweight Design Pattern

-- Design Patterns Series

with Mike Shah

19:00 - 20:00 CES (1pm ET) Tue. March 19, 2024

60 minutes + 15 minute Q&A After Introductory Audience

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

Thank you!

Extras and Notes