

MSc Computer Games and Entertainment

Maths & Graphics Unit 2011/12

Lecturer: Gareth Edwards

Diary For This Week

Diary For This Week

- **Wednesday 17th:**
 - From 11:00 to 13:00:
 - Fundamentals
 - From 13:30 to 17:30:
 - Introduction to Houdini
 - Programming for Vectors using C++ Operator Overloading
 - Introduction to Matrices
- **Thursday 18th:**
 - From 13:30 to 15:30 (and beyond?):
 - Houdini lecture by SideFX

Programming for Vectors using C++ Operator Overloading

Why Vectors?

- Programming for Vectors is a fundamental body of applied mathematical knowledge required for creating computer graphics software solutions in for Computer Games and Entertainment
- Used EVERYWHERE!

Why Operator Overloading?

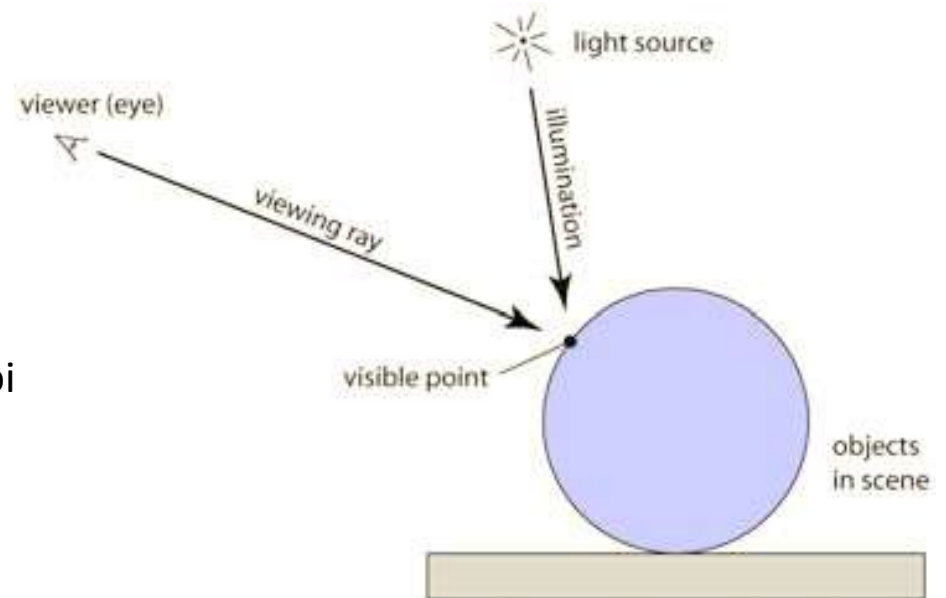
- Uses one of the really powerful features of the C++ language

Example: Used In Calculating Illumination

- Used in image synthesis for creating rendered 3d scenes depicting lit 3d objects, by ray casting, ray tracing, radiosity , etc.

Example ray tracing Psuedocode:

```
for each pixel {  
    compute viewing ray  
    intersect ray with scene  
    compute illumination at visible poi  
    put result into image  
}
```



Programming for Vectors using C++ Operator Overloading

Operator Overloading (1)

- One of the nice features of C++ is that you can give special meanings to operators, when they are used with user-defined classes.
- This is called *operator overloading*.
- You can implement C++ operator overloads by providing special member-functions on your classes that follow a particular naming convention.
- For example, to overload the + operator for your class, you would provide a member-function named operator+ on your class.

Assignment Operator =

- The assignment operator has a signature like this:

```
class MyClass {
    public:
        ...
        MyClass & operator=(const MyClass &rhs);
        ...
}
MyClass a, b;
...
b = a; // Same as b.operator=(a);
```

- Notice that the = operator takes a const-reference to the right hand side of the assignment. The reason for this should be obvious, since we don't want to change that value; we only want to change what's on the left hand side.

Operator Chaining (1)

- Also, you will notice that a reference is returned by the assignment operator.
- This is to allow **operator chaining**. You typically see it with primitive types, like this:

```
int a, b, c, d, e; a = b = c = d = e = 42;
```

- This is interpreted by the compiler as:

```
a = (b = (c = (d = (e = 42))));
```

Operator Chaining (2)

- In other words, assignment is **right-associative**.
- The last assignment operation is evaluated first, and is propagated leftward through the series of assignments.
- Specifically:
 - `e = 42` assigns 42 to `e`, then returns `e` as the result
 - The value of `e` is then assigned to `d`, and then `d` is returned as the result
 - The value of `d` is then assigned to `c`, and then `c` is returned as the result
 - etc.

Operator Chaining (3)

- In order to support operator chaining, the assignment operator must return some value.
- The value that should be returned is a reference to the *left-hand side* of the assignment

Const (1)

- Notice that the returned reference is *not* declared const.
- This can be a bit confusing, because it allows you to write crazy stuff like this:

```
MyClass a, b, c;  
...  
(a = b) = c; // What??
```

Const (2)

- At first glance, you might want to prevent situations like this, by having `operator=` return a `const` reference.
- However, *statements like this will work with primitive types.*
- Even worse, some tools actually rely on this behaviour.
- Therefore, it is important to return a **non-const** reference from your `operator=`.
- The rule of thumb is, "If it's good enough for `ints`, it's good enough for user-defined data-types."

Const (3)

- So, for the hypothetical MyClass assignment operator, you would do something like this:

```
// Take a const-reference to the right-hand side of the assignment.
// Return a non-const reference to the left-hand side.
MyClass& MyClass::operator=(const MyClass &rhs) {

    ... // Do the assignment operation!
    return *this; // Return a reference to myself.

}
```

- Remember, this is a pointer to the object that the member function is being called on.
- Since `a = b` is treated as `a.operator=(b)`, you can see why it makes sense to return the object that the function is called on; object `a` is the left-hand side.

Self Assignment (1)

- Now, one more **very important** point about the assignment operator:

YOU MUST CHECK FOR SELF-ASSIGNMENT!

Self Assignment (2)

- This is especially important when your class does its own memory allocation.
- Here is why: The typical sequence of operations within an assignment operator is usually something like this:

```
MyClass& MyClass::operator=(const MyClass &rhs) {  
    // 1. Deallocate any memory that MyClass is using internally  
    // 2. Allocate some memory to hold the contents of rhs  
    // 3. Copy the values from rhs into this instance  
    // 4. Return *this  
}
```

Self Assignment (3)

- Now, what happens when you do something like this:

```
MyClass mc;  
...  
mc = mc; // BLAMMO.
```

- You can hopefully see that this would wreak havoc on your program.
- Because `mc` is on the left-hand side *and* on the right-hand side, the first thing that happens is that `mc` releases any memory it holds internally.
- But, this is where the values were going to be copied from, since `mc` is also on the right-hand side! So, you can see that this completely messes up the rest of the assignment operator's internals.

Self Assignment (4)

- The easy way to avoid this is to **CHECK FOR SELF-ASSIGNMENT**.
- There are many ways to answer the question, "Are these two instances the same?"
- But, for our purposes, just compare the two objects' addresses.
 - If they are the same, then don't do assignment.
 - If they are different, then do the assignment.

Self Assignment (5)

- So, the correct and safe version of the MyClass assignment operator would be this:

```
MyClass& MyClass::operator=(const MyClass &rhs) {  
  
    // Check for self-assignment!  
    if (this == &rhs)          // Same object?  
        return *this;         // Yes, so skip assignment,  
                               and return *this.  
  
    ... // Deallocate, allocate new space, copy values...  
  
    return *this;  
}
```

Self Assignment (6)

- Or, you can simplify this a bit by doing :

```
MyClass& MyClass::operator=(const MyClass &rhs) {  
  
    // Only do assignment if RHS is a different object from this.  
    if (this != &rhs) {  
  
        ... // Deallocate, allocate new space, copy values...  
    }  
    return *this;  
}
```

Self Assignment (7)

- Remember that in the comparison, `this` is a pointer to the object being called, and `&rhs` is a pointer to the object being passed in as the argument. So, you can see that we avoid the dangers of self-assignment with this check.
- In summary, the guidelines for the assignment operator are:
 - Take a const-reference for the argument (the right-hand side of the assignment).
 - Return a reference to the left-hand side, to support safe and reasonable operator chaining. (Do this by returning `*this`.)
 - Check for self-assignment, by comparing the pointers (`this` to `&rhs`).

Compound Assignment Operators += -= *= (1)

- I discuss these before the arithmetic operators for a very specific reason.
- The important point is that these are *destructive* operators, because they update or replace the values on the left-hand side of the assignment. So, you write:

```
MyClass a, b;  
...  
a += b; // Same as a.operator +=(b)
```

Compound Assignment Operators `+=` `-=` `*=` (2)

- In this case, the values within `a` are *modified* by the `+=` operator.
- How those values are modified isn't very important - obviously, what `MyClass` represents will dictate what these operators mean.
- The member function signature for such an operator should be like this:

```
MyClass& MyClass::operator+=(const MyClass &rhs) {  
    ...  
}
```

- We have already covered the reason why `rhs` is a const-reference.
- The implementation of such an operation should also be straightforward.

Compound Assignment Operators += -= *= (3)

- But, you will notice that the operator returns a MyClass-reference, and a non-const one at that.
- This is so you can do things like this:

```
MyClass mc;  
...  
(mc += 5) += 3;
```

- Don't ask me why somebody would want to do this, but just like the normal assignment operator, this is allowed by the primitive data types. Our user-defined datatypes should match the same general characteristics of the primitive data types when it comes to operators, to make sure that everything works as expected.

Compound Assignment Operators += -= *= (4)

- This is very straightforward to do.
- Just write your compound assignment operator implementation, and return *this at the end, just like for the regular assignment operator.
- So, you would end up with something like this:

```
MyClass & MyClass::operator+=(const MyClass &rhs) {  
    ... // Do the compound assignment work.  
    return *this;  
}
```

Compound Assignment Operators += -= *= (5)

- One last note:
 - In general you should beware of self-assignment with compound assignment operators as well.

Binary Arithmetic Operators + - * (1)

- The binary arithmetic operators are interesting because they don't modify either operand - they actually return a new value from the two arguments. You might think this is going to be an annoying bit of extra work, but here is the secret:
 - **Define your binary arithmetic operators using your compound assignment operators.**

Binary Arithmetic Operators + - * (2)

- So, you have implemented your += operator, and now you want to implement the + operator.
- The function signature should be like this:

```
// Add this instance's value to other, and return a new instance
// with the result.
const MyClass MyClass::operator+(const MyClass &other) const {
    MyClass result = *this;    // Make a copy of myself.
                               // Same as MyClass result(*this);
    result += other;          // Use += to add other to the copy.
    return result;           // All done!
}
```

- Simple!

Binary Arithmetic Operators + - * (3)

- To summarise, the guidelines for the binary arithmetic operators are:
 - Implement the compound assignment operators from scratch, and then define the binary arithmetic operators in terms of the corresponding compound assignment operators.
 - Return a const instance, to prevent worthless and confusing assignment operations that shouldn't be allowed

Comparison Operators == and != (1)

- The comparison operators are very simple.
- Define == first, using a function signature like this:

```
bool MyClass::operator==(const MyClass &other) const {  
    ...  
    // Compare the values, and return a bool result.  
}
```

- The internals are very obvious and straightforward, and the `bool` return-value is also very obvious.

Comparison Operators == and != (2)

- The important point here is that the != operator can also be defined in terms of the == operator, and you should do this to save effort.
- You can do something like this:

```
bool MyClass::operator!=(const MyClass &other) const {  
    return !(*this == other);  
}
```

- That way you get to reuse the hard work you did on implementing your == operator.
- Also, your code is far less likely to exhibit inconsistencies between == and !=, since one is implemented in terms of the other.

Required For Next Week

Required - reading and work to be undertaken for next week....

So:

- Lengyl – Read Chapter 3 on Matrices again!
- Dunn & Parberry Chapter 7 on Matrices again!
- If you have already not done so – then compile and run RTVS_Lite.
- Compile and run RTVS_Lite_03 with Operator Overloading.
- Add vector arrow heads to all vectors in RTVS_Lite_03 .
- Skim through the course book “The Algorithmic Beauty of Plants”.

Course Lecture Notes, Resources and Books

Course materials are hosted – temporarily – on:

- <http://www.gjedwards.com/client/msc2012/>
 - 11_10_06_A_Lecture_01_Introduction.zip
 - 11_10_06_A_Lecture_01_Introduction.zip
 - 11_10_13_Lecture_02_Geometry.zip
 - 11_10_13_Lecture_03_Vectors.zip
 - Resources/
 - Book
 - CurvesSurfGfx-Salomon-2005
 - ELengyel_Mathematics_for_3D_Game_Programming_and_Computer_Graphics
 - F-Dunn-I-Parberry-3DMathPrimerForGraphicsAndGameDevelopment
 - The Algorithmic Beauty of Plants
 - RTVS_Lite.zip

End