Category	**Notes for Students**
Software Engineering	
Memory leaks	Be sure that any memory you allocate is later freed. Using smart pointers will make this much easier! See Basic Memory in Intro to C++
GPU memory leak	Every glGen*** call needs a glDelete*** call to avoid memory leaks on the GPU. See Lab 1.
Doesn't compile	Your code must compile on the department machines. Check that it does before handing in.
Crashes	TAs will test every edge case imaginable. Test them yourself so we don't have to take off for crashes!
Moderate to severe resource ownership issues	Clear ownership begets clear resource management. The object that creates a resource is responsible for it. See Resource Ownership in Intermediate C++.
Severe violation of Single Responsibility Principle - long methods, god classes, etc.	Following the Single Responsibility Principle will make refactoring and tracking down bugs exponentially easier. See the Software Engineering slides
Not using classes	Please use classes. Use structs for Plain Old Data. See Intro to C++ and Software Engineering
Code duplication	If you're repeating code within a class, make a function for it. See the Polymorphism and Composition section in the Software Engineering slides for other guidelines.
Not using inheritance in obvious situations	You should be creating (the C++ versions of) abstract classes and interfaces for classes that share functionality.
Public member variables	Always prefer private member variables with getters/setters.
Public implementation functions that should be private	Always prefer private member functions unless you are calling them from another class.
Bad/counterproductive variable and function names	Always use clear identifiers that make your intent obvious. This makes code much more readable.
Serious spaghetti code	Complicated if/else statements with lots of conditions are difficult to read. Wherever possible, simplify your logic.
Encapsulation violations (Liskov Subtitution Principle)	See Software Engineering slides.
Not using GLM	Don't reinvent the wheel. GLM provides tons of vector/matrix functionality.
Protected member variables in end-subclasses	If you are not going to make further subclasses, there is no reason to use protected variables. Make them private!
Neglecting to use the Standard Library	Don't reinvent the other wheel. std contains lots of functionality for math, strings, etc.
Magic numbers	Use clearly named constants close to where they are used. Capitalize constant names. e.g. const float SCALE_FACTOR = 2.5f;
Using multiple inheritance when composition would be better; using multiple inheritance as a replacement for Java interfaces	If there are better ways to do something, then don't use multiple inheritance.
C++	
Doesn't use initializer list / uses constructor body *instead*	By the time you get to the constructor body, the object should be usable. See Intro to C++.
Forgets a variable or 2 in init list	Don't always rely on default constructors. Initialize your member variables. See Intro to C++.
"init" functions	Use initializer lists. init functions are indicitive of bad practices. See Intro to C++.
Use of malloc/free	Use new/delete so that constructors and destructors are called. See Intro to C++.
Fails to use smart pointers in obvious cases	Prefer smart pointers because automatic memory management is great. See Intro to C++ and Intermediate C++.
Uses multiple layers of pointers when unnecessary (like a pointer to a smart pointer)	It's unnecessary. It also makes your code logic harder to read.
Fails to use automatic memory management data structures	Prefer standard data structures like vectors over dynamically allocated arrays. See Intro to C++.
Doesn't pre-allocate memory (resize, reserve) for a vector and instead repeatedly uses push_back.	It's inefficient - when the vector increases its capacity, it will have to copy over all the things inside it to a new vector.
Inefficient pass by value (not passing by reference for non-primitive types)	Repeatedly copying large data structures is incredibly inefficient. Use (const) references instead. See Intro to C++.
Returning a non-primitive member variable by value rather than by const-ref	See above.
Moderate to severe unwarranted "pass by pointer" rather than pass by (const) ref	(const) References are much safer and simpler than raw pointers. See Intro to C++.
C-style casting	This is C++, not C. C style casts can also be inefficient and unsafe. See Intermediate C++.
Improper use of virtual	Virtual = overridable in subclass. No subclass = no virtual. Virtual destructor = polymorphic class. Not polymorphic class = no virtual destructor. See Intro to C++.
Calls virtual function in base class constructor	This almost certainly does not do what you want it to do. Indicative of bad design. Also potentially unsafe
Initing/setting pointers to NULL or 0 unless explicitly tolerating OpenGL	Use nullptr. NULL/0 is an int. nullptr is a pointer type.
Includes in header file when forward declare will work better	See section 1.6 of Software Engineering on forward declaring.
"Safe deletes" i.e. zeroing/nullptr'ing pointers after delete	Indicative of bad design - resource management is unclear, consistency is semi-broken. Should be using smart pointers anyway.
Unnecessary scope resolution	e.g. void Foo::func(){ Foo::otherFunc(); }
Being naughty with the preprocessor	Defining constants and/or function macros is very bad practice. Use C++ instead
Pointer-to-pointer 2D arrays	Use a 1D array and index into it. Don't allocate pointers to pointers.

Forward declaring / "tampering" with std namespace	Always just include standard library headers. This overrides the usual forward-declaration pattern
Implicit primitive type conversions	e.g. int x = 5.5f; or float y = 2; Usually this will be fine, but be aware of it. (No points off)
Not using source files (combined souce and header files)	The only exception is template programming, which CS123 is not doing
Non-member variables in .cpp file	Breaks encapsulation, most likely won't do what you want it to. Make it a member variable or static class variable.
Non-member functions in .cpp file that *should* be member functions	No free-floating functions
Improper use of types (e.g. passing int instead of bool)	C++ has bool. Use it. (No points off.)
Making enums instead of enum classes	Enum classes are the new and better versions of enums. Using existing enums is of course fine. (No points off)
const_cast	Breaks explicitly specified program invariants. Come to TA hours if you feel justified in using it.
auto_ptr	auto_ptr is deprecated. Use unique_ptr. (No points off)
goto	Don't.
OpenGL	
Use of clearly deprecated functionality we don't teach (glBegin, glPushMatrix, glAccum, etc.)	Deprecated.
GL_QUADS	Deprecated.