* Important concepts
  + Encapsulation
  + Inheritance
  + Polymorphism
* Finite State Machines
* Precompiled headers
* Exception-safe coding
  + Only use new/delete inside constructors/destructors.
* Singleton vs. namespace + free functions
* Designing for cross-compatibility
  + SDL2, OpenGL, and libraries based on them abstract away all the platform specific code.
* Virtual Inheritance and the diamond problem
* C++11
  + New bracket initializer lists avoid creation extra object and calling of copy constructor.
  + Initialize non-static members in class declaration, instead of in constructor.
  + Inheriting constructors
  + enum class: doesn’t act like an int
* Const: When it shouldn’t be ignored and when it is unnecessarily reducing readability.
  + bitwise constness (physical constness) vs. logical constness
* Optimize data for cache usage (and the pitfalls of OOP)
  + Keep data in contiguous memory in the order that you process it.
    - Keep particle array sorted by whether it is active, so that when you process the whole array, you only load things into the cache that you are actually going to use.
      * Checking a particle for an active flag loads all of its data into cache which is a waste if it’s inactive.
      * Keeping all active particles in the front allows you to remove from the loop the control flow structure that checks whether the particle is active, improving the ability of the pipelining/branch prediction capability of the CPU to speed things up.
        + Activate particle: swap it with the first inactive particle, and increment the active particle count.
        + Deactivate particle: swap it with the last active particle, and decrement the active particle count.
      * Moving things around in memory is preferable to the cache misses or branch mispredictions you’d otherwise get in your update loop.
      * Suggestion: Keep your game object components in regular arrays. Have an array for each type of game component.
  + Hot/cold splitting
    - Give the hot a pointer to the cold.
      * Hot data needs to be touched every frame.
      * Cold data gets used less frequently and doesn’t get loaded into cache with its owner unless it’s requested specifically.
    - It’s often difficult to decide what’s hot or cold.
    - Don’t get sucked into spending endless time pushing data around.
* Text alignment: origin when dtx starts drawing = lower left corner of the first character of the first line.
  + Had to figure this out on my own with testing.
  + For my engine, the text position can represent any of nine anchor points on the resulting text box instead of just one.
    - This seems more useful than specifying the bottom-left of the first line, although I could also add that as a tenth anchor point.
* O(log N)
  + Half of the input is discarded in each step.
  + Doubling of the input size results in one more step.
* Creating a resource manager.
  + vector, map, set, unordered map?
  + smart pointers, or raw pointers with manual reference counting
    - duplicate assets in memory if the resource manager releases
  + A Simple Fast Resource Manager using C++ and STL
    - Loading and accessing everything that’s loaded from files before use.
    - Fast and efficient, uses minimum memory, easy to use.
    - Approach:
      * templates
      * base resource class
      * reference counting
      * STL
    - Structure
      * Resource
        + handle
        + reference count
        + path and name of file
* Smart pointers
  + Maintaining class invariants is the essential function of shared\_ptr
  + Using shared\_ptr easy decoupled destruction-notification (no *real* ownership)
  + shared\_ptr vs. unique\_ptr
    - Not a question of *efficiency*, but *semantics*.
      * Uniqueness: when you read the code it’s very clear the lifetime and ownership of the object.
    - unique\_ptr is safer (no circular references) and just a little bit faster.
      * unique\_ptr has no overhead compared to a raw pointer, but it has advantages.
    - Only use shared\_ptr if you need shared ownership.
    - Prefer values > unique\_ptr > shared\_ptr
    - Some say new and delete are deprecated.
* I just realized that the new C++11 range-based for-loop will allow me to make a lot of my code look much cleaner!
  + It’ll replace all those iterators, and make things much easier.
  + I am going to go through the code I’ve written so far and fix it up.
* I should read about writing secure code.
  + Preventing buffer overflow and double free attacks.
* Pre-increment and pre-decrement are preferable. Post-increment and post-decrement waste instructions.
  + Most important for complex types like iterators.
* If you have lots of small game assets each under 4 kilobytes, pack them together into a single file, a sprite-sheet, or packed into the executable.
* Don't put 'using' statements at file scope in header files, as they pollute whatever scope the file is included in.
  + Just get used to the namespace prefixes.
* I changed Renderer to be a namespace with static functions (basically like a static class but better because the private variables aren’t exposed with the interface)
  + I have sacrificed explicitness.
    - When the Renderer was passed around as a parameter, it was very clear which code has access to the Renderer.
    - But after the change, any code may access the Renderer, which in general can end up making things more confusing if, for example, you’re collaborating with other programmers on the project.
    - I should either self-enforce a rule that says: Only code that’s in a function name Draw…() is allowed to call Renderer::Draw…() functions.
      * Or I could switch back to using a regular class and passing a Renderer pointer around, which would make interfaces more self-documenting.
        + I think this is probably the proper way of doing it.
        + On the other hand, I like having all rendering go through one renderer. If I ever need multiple windows I can simply extend the Renderer to support multiple windows just like it supports multiple fonts and textures by using resource managers.
  + I tried going back to a regular class, but I quickly reverted to namespace functions because I couldn’t ensure that the same Renderer would be passed into the GameState ctor and Draw function.
    - If I need to add multi-window support I can use resource IDs just like textures/fonts.
  + OpenGL is set up this way with functions, and it seems like mimicking that idea would be a safe thing to do as I’m just starting out with the engine.
* Aspect Ratios (for space game): Have a default distance from the player to the corner of the screen, and allow the player to zoom in or out from there.
* The reason the class (with an interface) is more encapsulated than the struct (with exposed data) is that more code might be broken if the (public) data members in the struct change than if the (private) data members of the class change.
* More member functions means less encapsulation, so prefer non-member non-friend functions over member functions.
* Microthreads implemented in assembly is a platform-independent way to make Game AI faster and are.
* Store elapsed game time in a double, starting at 2^32 or any large power of 2, to get constant precision for 136 years of game running time.
* Interfaces need to make the ease of executing any task proportional to the frequency of the task multiplied by its importance
* Minimize reliance on internal states.
  + If you have two equally complex ways to implement a feature, prefer the one that solely relies on the input it was given and returns a value over the one that relies on internal states and modifying them.
    - simpler testing, easier code reuse, less surprises
* Identifying modules:
  + Separate parts you want to reuse from the specifics of the game.
  + It may seem like more work at first, but you’ll develop a reusable codebase that will make later work easier.
  + If you find yourself writing similar code (especially if you’re copying and modifying a bunch of code
  + For any specific piece of reusable code, it will take at least three uses before you get the interfaces right.
* Menu( Background, logo, menu );
* Engine:
  + Separation of Gameplay
    - The engine “includes” the gameplay library (flipping the idea of a script on its head).
    - The designers and gameplay developers create the world and engine developers use this world.
    - The engine should communicate with the gameplay library through interfaces.
    - Both sets of developers, engine and gameplay, have set functionality they must achieve, a simple, well defined interface, and are freed up to work at full velocity.
* You have to create and set up separate VAOs for each shared gl context.
  + The vertex buffer will be shared across contexts, but not VAOs.
* Use regular local variables, not static variables, inside functions.
  + Moving the stack pointer is much quicker than accessing static variable.
* Text alignment system is not 100% perfect.
  + I could switch libdrawtext to some other library, or do TrueType fonts myself.
* Temporary variables can be used to clean up the code, making it easier to read and understand, and can save you from having really long lines of code.
* Effective Modern C++
  + \_\_\_Template type deduction
  + \_\_\_**auto** type deduction
  + \_\_\_decltype
  + Prefer **auto** to explicit types.
  + \_\_\_View deduced types in IDE
* Pong
  + 11 points to win
  + width is double height
  + paddles 1/5 height
  + paddle speed limit ¼ initial ball speed
  + the vertical component of the velocity is increased by two percent every time the ball hits a paddle
    - (With a limited paddle speed based on the initial velocity of the ball, the ball eventually travels too fast for the paddle to reach it)



