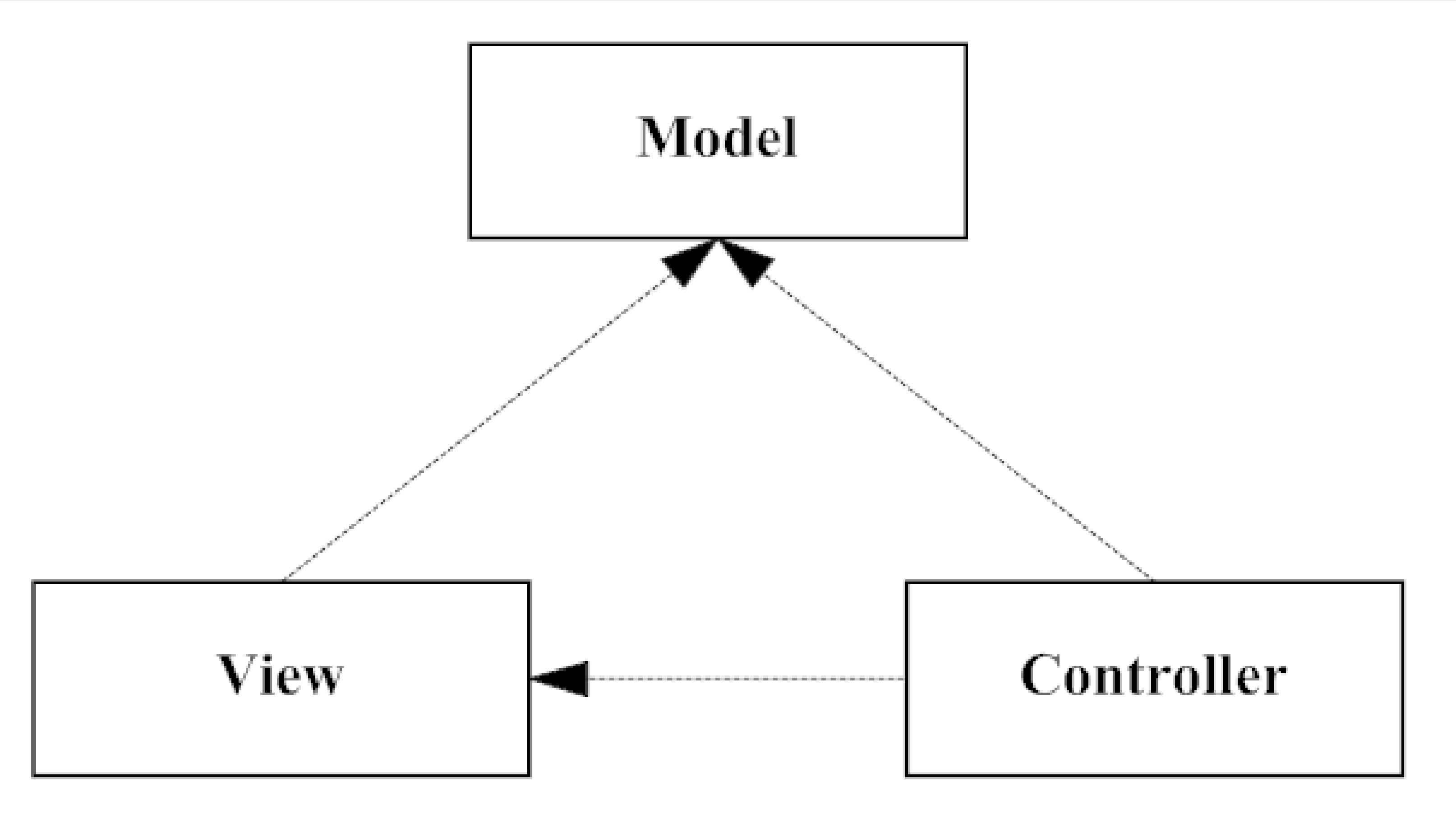
Model / View / Controller



MVC origins

- 1979 - Trygve Reenskaug
  - working in Smalltalk at Xerox PARC
  - paper: Applications Programming in SmallTalk-80 : How to use Model-View-Controller
  - “isolates business logic from user interface considerations”

MVC concepts

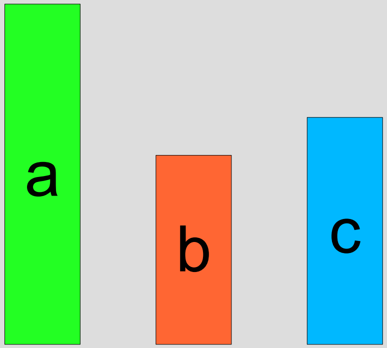
- Model – application state and business logic
- View – user interface / visualization
- Controller – manages communication of user actions to Model

MVC example

Model

int a, b, c;

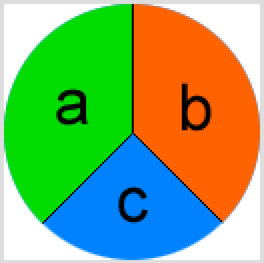
BarChartView



SpreadSheetView

a	7
b	4
c	5

PieChartView



Potentially many views!

Microsoft Foundation Classes

- MFC
  - User interface framework for creating applications with a common look and feel (i.e. Windows!)
  - MFC Document / View
    - Document = “Model”
    - View = “View and Controller”, handling both user “input” and visualization “output”

Implementation

- Classic MVC as well as MFC Document / View assume that Views are **stateful**
- Views encapsulate state and behaviour = OOP objects, instances of classes

Implementation

- Stateful Views brings up the issue of **synchronization**
- Views need to synchronize own state to Model state
- “Gang of Four” (Design Patterns) describe various tradeoffs in the Observer pattern...

MVC is a good thing!

- Generally very applicable to a wide range of application domains
- Clearly separates concerns:
  - Model = state & business logic
  - View = gui and presentation
  - Controller = routes user input to changes in Model

HOWEVER!

While the premise of a View being **stateful** is intuitive for programmers trained in OOP (i.e. an object encapsulates state and behaviour), this can be **severely limiting!**

The Issue & The Assumption

- The main issue:
  - Views implicitly cache Model state
- The assumption:
  - A visualization should be / is **stateful...**

The Issue & The Assumption

- The main issue:
  - Views implicitly cache Model state
- The assumption:
  - A visualization should be / is **stateful...**

FALSE!

“but we've always done things this way!”

- The premise of a stateful View is DEEPLY entrenched in a wide range of popular object oriented visualization / gui packages:
  - MFC
  - Ogre
  - HTML DOM
  - Java Server Pages / Java Beans

Trends in the Swedish game industry

- I have never seen a professional (Swedish) game renderer that is NOT stateful!
- This is called **Retained Mode** (i.e. the renderer "retains" application state internally)
- This is the classic "scene graph", a hierarchy of "nodes"
- node = mesh + material + transform + etc

Why Retained Mode?

- Historically, Retained Mode was required to achieve real-time performance
  - i.e. recursive frustum culling of pre-transformed bounding hierarchies
  - “retain much state and only update when absolutely required”

Windows too!

- The design of Win32 / GDI based on the same principle
- Ca 1993 too expensive to repaint hi-res displays (640x480x8bit) at 60hz (multiple overlapping windows, etc)
- GDI operates asynchronously with a system of “dirty rects” and centrally managed rendering, i.e. WM_PAINT messages

MFC as a wrapper

- MFC = an OOP wrapper for Win32 / GDI
- Exposes aspects of the gui as classes / objects, with interfaces to allow transfer of application state to gui state
- Rendering not controlled by application, i.e. “The Hollywood Principle = don't call us, we'll call you...”

Direct X

- Developed in an attempt to make Windows 95 a viable platform for games (i.e. GDI is too slow!)
- DirectX Graphics (ca 1996) had both Retained Mode (RM) and Immediate Mode (IM)

Direct X Immediate Mode

- No “scene graph”, no “instance” abstraction, no “camera” abstraction
- Little retained state
- Low level
- Complex
- Application needs to do lots more work, but game coders wanted the control... RM soon dropped from DirectX

Early trends

- Games tended to build a “renderer” on top of the low level DirectX IM interfaces
- This was before HW T & L!
- Games needed to be very smart about what they sent to the GPU
- Ca 1997, Quake 2 used BSP / PVS to minimize the amount of geometry sent to the GPU

Early trends

- Standardization of common tasks, need for productivity, asset pipeline issues... all led to the “engine” approach
- Low level API's wrapped into a higher level abstraction that the application level programmers could use more easily
- Also, we all loved OOP and Design Patterns...

Today

- GPUs are outperforming CPUs in number-crunching applications
- The biggest bottleneck is the bus
- Brute force approaches are starting to become more viable than clever CPU level optimizations
- The GPU can take it!

Jungle Peak ca 2006

- DirectX 9 HLSL application
- Sort by shader/texture/technique
- Huge batches (800 000+ vertices), single DrawIndexedPrimitive() call
- Merge multiple instances of meshes into single mesh, sorted by shader (i.e. forests)
- Did not split and cull to frustum, fewer draw calls gave us better performance

Conclusion

There is no longer any performance reason to have a stateful visualisation...

MVC revisited

- What is a non-stateful View?

MVC revisited

- What is a non-stateful View?
- Basically a procedural interface
- Very much what DirectX 9 is

MVC revisited

“Oh yeah? What about SetRenderState()?”

MVC revisited

- SetRenderState() basically avoids HUGE parameter lists in DrawPrimitiveXXX() methods
- Drivers reserve the right to propagate DirectX state to the GPU at **any time**, even waiting until the actual draw call

MVC revisited

- Observe also the design of HLSL
- Pushes the “stateless” concept even further, by supplying all state to the shader at render time

MVC revisited

“So, if View is 'a bunch of functions' with no state, how do we describe what gets rendered, and how, and when?”

MVC revisited

- Enter the **Controller**
- 2 jobs:
  - 1) doInput() = react to user input, and direct how that input is allowed to change Model
  - 2) doOutput() = dynamically, in real time, compose the current “view” of the application using View

MVC revisited

- Controller basically “programs” View to present a visualization to the user (in real time)
- This includes **everything** you see on the screen, including guis

MVC summary

- **Controller** manages both input and output
- **View** exposes ways to query user input, as well as render output, but is in itself entirely passive
- **Model** encapsulates application state as well as application logic / behaviour

MVC gains

- No more state caches
  - Removal of most (if not all) state caches in View make sync with Model trivial (if not non-existent)
  - Such (manual) syncing is often a major burden and source of bugs

MVC gains

- Dynamic “views”
  - Controller dynamically and procedurally “calculates” the “view” of the application.
  - Allows for any number of very disparate “views” and makes switching between them very easy to implement.
  - Allows for “in-app” editors, debug views, etc

Supporting technologies

- Immediate Mode Graphical User Interface (IMGUI)
  - New way of authoring and deploying guis
  - Much faster, more intuitive, and more productive than traditional Retained Mode guis (i.e. MFC)
  - Geared towards real-time applications with real-time rendering, i.e. perfect for games

Supporting technologies

- Automated Persistence
  - Basically a language extension for C++
  - Mark any data as persistent across application executions
  - Persistence is automatic and transparent

Tell me more!

IMGUI? Automated Persistence?

I'm working on the book...

http://www.johno.se