Demo Engine Tricks of the Trade

Boosting Productivity & Performance, Reducing Complexity

Armin Jahanpanah
spike / science

NVScene 2014

DEMOS = CREATIVITY + WORK^N

What we need

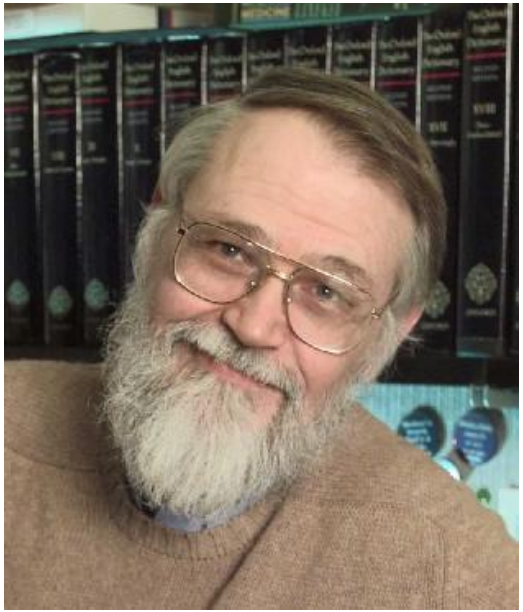
- Improved workflows, more productivity
  - Shorter turnaround times
  - Quicker/more iterations
  - Better end result
- Better Code
  - Avoid over-engineering, reduce complexity
    - Less and simpler code, more robust
  - Better performance

Agenda

- Code Complexity
- Workflow: Iteration & Tweaking
- DX11
- Performance

Part 1:

COMPLEXITY



“Controlling complexity is the essence of computer programming.”

- Brian Kernighan, *Software Tools* (1976)

The Golden Rule

- KISS: Keep It Simple & Stupid
- Avoid over-engineering, reduce complexity

„Debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

- Brian Kernighan

How?

- Focus on the task at hand
  - Don‘t write over-general systems („I _might_ need this later“)
  - Get the job done
    - But this doesn‘t mean you should write sloppy/ugly code
- Less code → reduced surface area / probability of bugs
- Avoid overly complex, *under-the-hood machinery (e.g. smart pointers)*
- Consider C instead of C++
- Simple, robust code that works and solves the problem
  - May not be fancy, super-general or OOP-guru-style
  - But it will safe you next time you’re trying to finish your prod 15mins before the deadline
    - Sleep-deprived, noisy environment, rushing

Part 2:

WORKFLOW

Asset Hot-Reloading

- Huge productivity boost for tweaking/iterating:
  - „Live“ shader editing while demo keeps running
  - Modify texture in Photoshop, save, instant update in engine
  - Adjusting meshes / scenes on-the-fly
  - …
- Easily implemented
- … yet few people actually do it!

Asset Hot-Reloading

- Some options:
  - Text file containing list of filenames, reload all on keypress
    - Pro: Doesn't need to reload all assets
    - Con: Need to manually update list of filenames
  - *Resource manager, periodic timestamp checks*
  - Full directory watcher using OS file notification mechanisms
    - Some amount work…
  - …

Example

Asset Hot-Reloading

Starting point:

A base class for all derived resource types

```
struct Resource
{
    virtual void Create(void* data, UINT size) = 0;

    char filename[256];
    time_t lastTime; // from <time.h>
};
```

```
struct Texture : public Resource
{
    Texture() : tex(NULL) {}
    ~Texture();

    void Create(void* data, UINT size);
    // via D3DXCreateTextureFromFileInMemory etc.

    IDirect3DBaseTexture9* tex;
};
```

Simple resource cache implementation

```
Resource* resources[256];
int numResources;

void LoadResource(Resource* resource, const char* filename) // private
{
    void* data = NULL;
    UINT  size = 0;
    struct _stat st;

    if ( _stat(filename, &st) == 0 ) {
        memcpy(resource->filename, filename, strlen(filename)+1);
        resource->lastTime = st.st_mtime;

        data = LoadFile(filename, size);

        resource->Create(data, size);
        resources[numResources++] = resource;

        delete[] data;
    }
}
```

```
// public
Texture* LoadTexture(const char* filename)
{
    Texture* tex = new Texture;
    LoadResource(tex, filename);
    return tex;
}
```

Reloading

- Check timestamp of each resource in list
  - Every Nth tick in your mainloop
  - Every N ms
  - Upon keypress
  - …

```
void ReloadResources()
{
    void* data = NULL;
    UINT  size = 0;
    struct _stat st;

    for (int i=0; i < numResources; i++)
    {
        Resource* resource = resources[i];
        if ( _stat(resource->filename, &st) == 0 ) {
          if (st.st_mtime != resource->lastTime) {
            resource->lastTime = st.st_mtime;

            data = LoadFile(filename, size);

            resource->Create(data, size);

            delete[] data;
          }
        }
    }
}
```

Nice and easy.

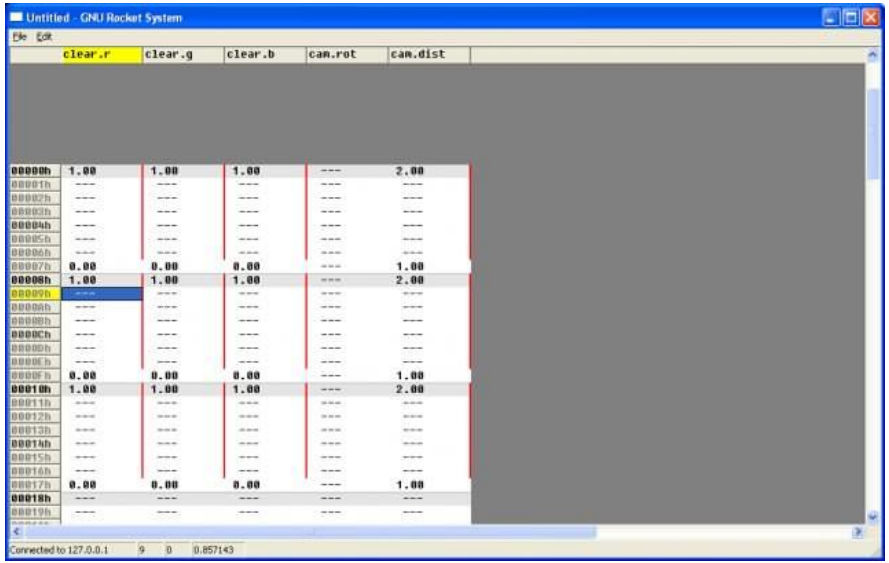
But we can take the concept further…

On-the-fly Tweaking

- Can use the same concept for config files
  - Keep effect variables in external text file, hot-reload
- Console, telnet(!)
- Or just use the *GNU Rocket System*!!

GNU Rocket System

- By Kusma & Skrebbel
- Standalone tool that connects to your demo via sockets API
- Tracker-like interface for syncing / key-framing



GNU Rocket System

- Download:
  - ***rocket***.*sourceforge.net*
  - *https://github.com/kusma/**rocket***
- Great tutorial by Gloom
  - *http://www.**displayhack**.org/2011/syncing-your-real-time-graphics-right/*

Part 3:

DX11

DX11 Resource Management

- Compared to DX9 the DX11 API introduces a lot of new objects to keep track of:
  1. *State objects* for blending, rasterization, sampling, etc.
  2. *Constant buffers* everywhere
  3. *Vertex layouts* are now more closely tied to shaders
     - Need to pass shader blob to `CreateInputLayout()`
- Goal: Abstracting it away
  - Avoid exposing all the details (create/release/etc.)

Constant Buffers

- Simple manager
- We can extend this approach and completely eliminate cbuffer objects from our engine‘s API
  - Just need Map/Unmap-style functions

```
std::vector<ID3D11Buffer*> m_ConstantBufferCache;

ID3D11Buffer* GetConstantBuffer(size_t sizeBytes)
{
    D3D11_BUFFER_DESC cbDesc;

    // first try to find already existing buffer with matching size
    for (size_t i=0; i < m_ConstantBufferCache.size(); i++)
    {
        m_ConstantBufferCache[i]->GetDesc(&cbDesc);
        if (cbDesc.ByteWidth == sizeBytes) return m_ConstantBufferCache[i];
    }

    // not found, we need to create a new one
    cbDesc.ByteWidth           = sizeBytes;
    cbDesc.Usage               = D3D11_USAGE_DYNAMIC;
    cbDesc.BindFlags           = D3D11_BIND_CONSTANT_BUFFER;
    cbDesc.CPUAccessFlags      = D3D11_CPU_ACCESS_WRITE;
    cbDesc.MiscFlags           = 0;
    cbDesc.StructureByteStride = 0;

    ID3D11Buffer* buffer = NULL;
    HRESULT hr = gpu.m_d3dDevice->CreateBuffer(&cbDesc, NULL, &buffer);

    // add to cache
    m_ConstantBufferCache.push_back(buffer);

    return buffer;
}
```

Vertex Layouts

- Different shaders usually require different vertex formats / layouts
  - 2nd UV set (lightmaps), Lighting: Normal/TS, Bone Weights/Indices, …
- Solutions:
  - Use a single (compressed) vertex format that contains everything
    - Should be 32 bytes, adds decoding overhead to vertexshader
      - Normals: DXGI_FORMAT_R8G8B8A8_SNORM, R10G10B10A2, 3 halfs, etc.
      - Lightmap Uvs can use 2 shorts
  - Use a small, fixed set of vertex formats
  - *Automatically create input layout directly from vertex shader code via D3DReflect API*
    - Can iterate over all vertex elements and deduce their type/format from desc
    - Implement a caching scheme by hashing the `D3D11_INPUT_ELEMENT_DESC`

Part 4:

PERFORMANCE

Boosting Performance

- Every API call has a certain CPU cost in the driver
- If you want to display a lot of (animated) objects you might quickly run into trouble
  - Debris, swarms, particles, etc.
- Subject of lots of buzz recently
  - That „Mental“ API
  - Bindless OpenGL – NVidia extensions
- *Two main approaches* to reduce API/driver overhead:
  - Reduce # of API calls
    - Example here: DX11 Constant Buffers
  - Make draw calls do more stuff – Instancing

Example: DX11 Constant Buffers

- Tip: Don’t create lots of cbuffers!
  - Referencing hundreds of different cbuffers per frame can induce substantial overhead
  - Only allocate one underlying D3D cbuffer per size class
    - Mapping (DISCARD) the *same* buffer 1000 times is much faster than mapping 1000 *different* buffers
- More details: See ryg‘s blog at http://fgiesen.wordpress.com/2013/03/05/mopping-up/

The Beauty of 90‘s GL: Display Lists

- As old as the hills
- Super easy to use: `glGenLists`, `glNewList`, lots of funky `glVertex3f` immediate mode madness, `glEndList`
- Just one API call per draw! `glCallList()`
  - Rivals VBO performance!
  - Driver optimizes data heavily
- Additional advantages:
  - Makes it easy to convert from facelist-based geometry data typically provided by DCC apps to GPU-friendly vertex stream
    - DDC apps typically store a list of faces that reference vertex positions, normals, Uvs via indexing into separate coordinate arrays (see also .OBJ)
    - With vertex buffers you usually need to untangle everything and duplicate vertex elements to yield a flat data stream
    - To prevent unnecessary vertex duplication, one has to implement a condensation/caching scheme
  - Vertex data condensation might be handled by driver

Instancing

Main concept: Upload single mesh, *single* call renders it *multiple* times according to data provided in separate vertex stream

GL Instancing

- VBO for mesh geometry as usual
- Create a VBO with `GL_ARRAY_BUFFER_ARB` and `GL_DYNAMIC_DRAW_ARB` that will be filled with *instancing data*
- Declare attributes of per-instance data in vertex shader
- Use `glVertexAttribPointer()` and `glVertexAttribDivisor()` to setup vertex streams
- Map and update instance data buffer
- Render everything in one go via `glDrawElementsInstanced()`
- See also „Instancing in OpenGL“ – *Jari Komppa, available online: http://sol.gfxile.net/instancing.html*

D3D Example: Momentous

- D3D10 version of the particle system used in "fr-059: momentum"

- Full source code at: https://github.com/rygorous/momentous

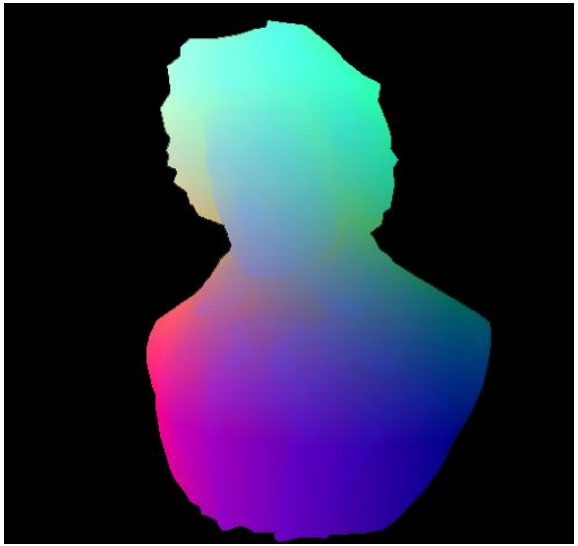
Material System

- The number of rendering modes and shading styles can quickly lead to an exploding number of possible combinations
- **Ubershader** concept can help to reduce/manage complexity
  - But requires implementation of a shader cache system at a certain point
- **Brute force**: Big, complex shaders where portions get filtered out by setting black/white textures or color constants
  - Can work well, if you just need a 3dsmax-like material concept (i.e. shading model + a number of textures and colors to tweak)

Deferred Shading Overview

- Deferred Shading approaches are more flexible
  - Write material values into framebuffer, plus XYZN (G-Buffer pass, MRT)
  - Lighting pass for each light:
    - Perform lighting computation for each pixel and add into accumulation buffer
  - Final composition pass combines lighting values with material coeffs and colors

Deferred Shading Buffers



Position

Normals

Lighting, Shading

Images by Carsten ‚Frenetic‘ Dachsbacher

Deferred Shading: Challenges

- Many (fullscreen) passes, fat RTs → Requires a lot of bandwidth
- Most complexity is in lighting pass
  - Diffuse/specular lighting model, attenuation, shadow mapping, etc.
- So instead of running over all pixels of the framebuffer, you want to only process pixels affected by the light
  - Screen-space scissor rectangle
  - Rendering geometry that approximates the bounding volume of the light (sphere for omni)
  - Marking affected pixels in the stencil buffer

Deferred Shading: KISS

- Just ignore the all the fancy/complex stuff!
  - Typical demos have few (1) light sources per scene
    - Definitely not hundreds/thousands
- Implementing a basic version is rather easy
  - Use world-space positions & normals
  - Simple lighting pass
    - No shadows
    - No scissoring/stenciling optimizations
- Result: We get all the nice properties with minimum effort!

Thank you!

Questions?

References

- Efficient Buffer Management - *John McDonald (NVidia)*
- Beyond Porting: How Modern OpenGL can Radically Reduce Driver Overhead
  - *Cass Everitt, John McDonald (NVidia)*
- OpenGL Performance Tuning
  - *Evan Hart (ATI), GDC 2006 [re display lists]*
- ATI OpenGL Programming and Optimization Guide
  - *[re display lists]*

References

- Inside Geometry Instancing – *Francesco Carucci, GPU Gems 2, available online (developer.nvidia.com)*
- Porting Soure to Linux
  - Rich Geldreich (Valve), John McDonald (NVIDIA), GTC 2013
- *Various talks at GTC 2014!*