Preparing to DX10: Advanced Rendering Effects

Guennadi Riguer

Outline

- DirectX 10 overview
- New pipeline and features
- Effect ideas and DirectX9 fallbacks

DirectX 10 Overview – Timeframe

- DirectX 10 requires Vista OS
  - ...and DX10-capable graphics card!
- DirectX 10 release alongside Vista
  - Currently planned for Q1 2007
- Beta program available for Vista and DX10
  - Register with Microsoft (connect.microsoft.com)
- DX10 SDK publicly available
  - With full REF implementation

DirectX 10 Design Goals

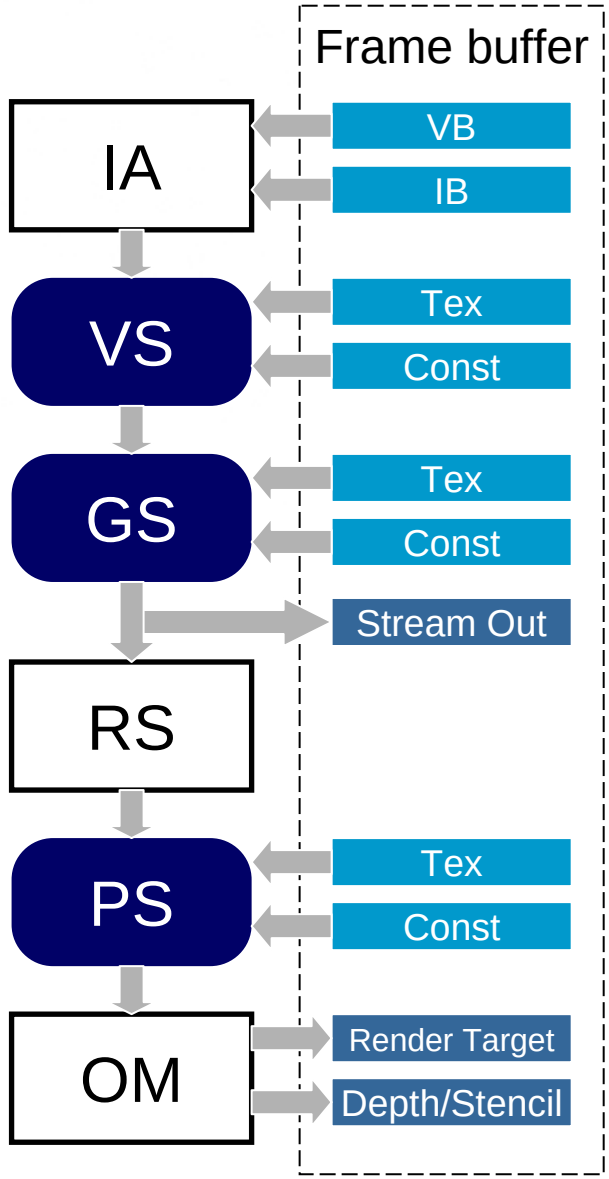
- Performance improvements
  - Enhancements for both CPU and GPU
- Advanced forward-looking feature set
  - No more fixed-function
  - No other unpopular legacy features
- Common feature set for all HW
  - No more caps!

Performance Improvements

- Major problem for ISV
- CPU bottleneck – “Small batch problem” addressed:
  - New driver model and thin runtime
  - Better state management
- API features to help performance
  - Instancing as “first class citizen”
  - Resource management and data recirculation
  - Render more in fewer draw calls

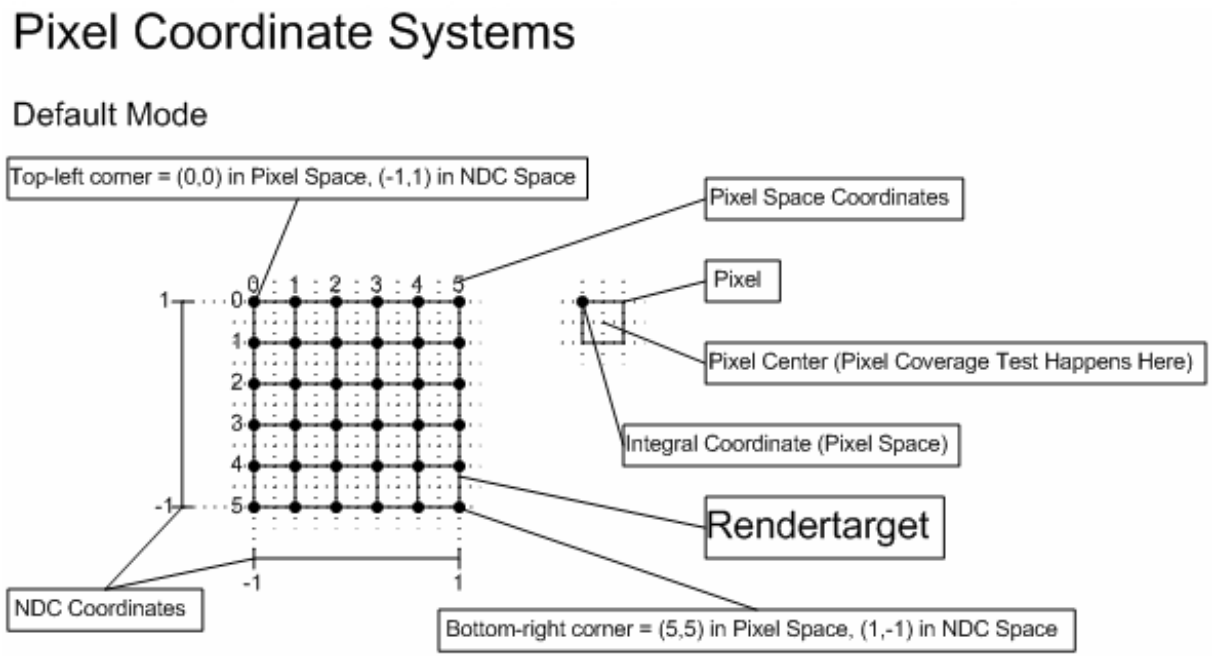
New Pipeline

- Orthogonalized FB
- Input assembler (IA)
- Geometry shader (GS)
- Stream out (SO)
- Output merger (OM)



Pixel Coordinates

- New pixel coordinates rules for D3D10!
- Pixel centers now offset by (0.5f, 0.5f)
  - Similar to OpenGL rules



State Objects

- State Objects replace renderstates
- Can be stored in video memory for better efficiency
- Five different state objects available:
  - Input Layout Object
  - Rasterizer Object
  - DepthStencil Object
  - Blend Object
  - Sampler Object

Orthogonalized FB

- All resources are just memory arrays
- Multiple “views” provide attachment points to different stages in the pipeline
- Allows recirculation of data
- Texture fetch in VS, GS and PS

Orthogonalized FB

- Can re-interpret some resource data
- Not fully general memory access
  - Would hurt HW performance
  - E.g. can’t render to constant buffer, etc.

Resource Types

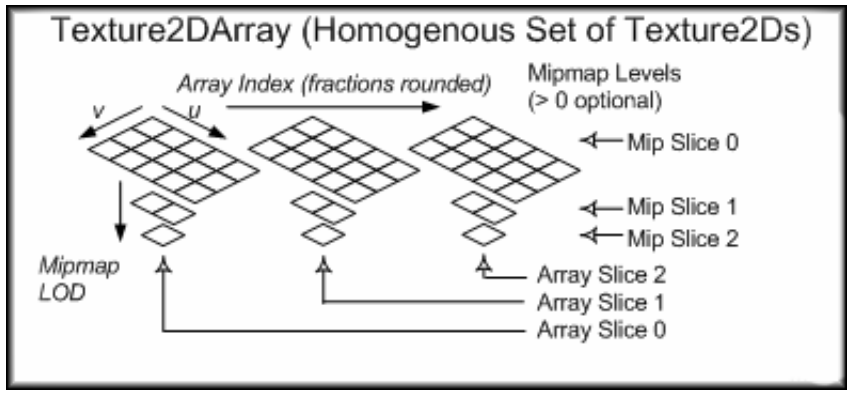
- Two resource types:
  - Buffer
  - Texture
- Allows distinction required when filtering or streaming out
- Texture resources made up of subresources
  - MIP level or array of MIP levels for arrays

Resources and Views

- All resources are memory arrays
- Allows reinterpretation of video memory through “views”
- A resource view defines how a resource is accessed
  - Example 1: a specific MIP level can be selected as a render target
  - Example 2: 3D texture viewed as an array of 2D render target textures

Texture Arrays

- Up to 512 texture slices per array
  - Up to 128 textures/arrays can be bound to a pixel shader
- Allows sampling of a texture slice based on a dynamic index
  - Allows material lookup per ID



New Cool Things You Can Do

- More efficient multi-pass rendering
  - E.g. animate once, render many times
- Displacement mapping
- Neat tricks with re-interpreting data
- GPGPU driven effects

Predicated Rendering

- Occlusion based rendering decision without CPU intervention
  - Render simple proxy geometry
  - If not visible, rendering of more complex geometry will be skipped
- Unavailable query results are treated as “possibly visible” (conservative test)
  - Predicated test should be sent well before the subsequent draw
- DON’T use this as an alternative to view frustum culling on the CPU

Constant Buffers

- Much bigger constant store
  - 15 arrays of 4096 elements each
- Faster constant updates
- Ability to group consts into multiple buffers for more efficient updates
- No more DX9 limits!

Constant Buffers

- Constant buffers declared within a namespace

```
cbuffer MyConstantBlock
{
    float4x4 matMVP;
    float3 fLightPosition;
}
```

- Group constants into buffers according to update frequency
- Constant buffers updated with resource loading operations
  - Map()/Unmap() or UpdateSubResource()

Geometry Instancing

- Allows multiple objects to be rendered in a single call
  - With varying material properties
- Use texture arrays to change textures on instances
- Core feature of DirectX 10

  ID3DDevice::IASetVertexBuffers()

  ID3DDevice::DrawInstanced(...)

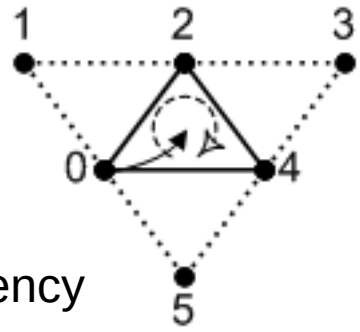
  ID3DDevice::DrawIndexedInstanced(...)

System Generated Values

- InstanceID (VS)
  - An integer ID for the object instance being processed
- VertexID (VS)
  - An integer ID for the vertex being processed
  - Indexed primitives: VertexID contains index number
- PrimitiveID (GS, or PS if GS not bound)
  - An integer ID for the primitive being processed

Geometry Shader

- Operates on primitives
- Can change topology
  - Generate new primitives
  - Remove existing primitives
- Can have access to adjacency information



Triangle with adjacency

Stream Out

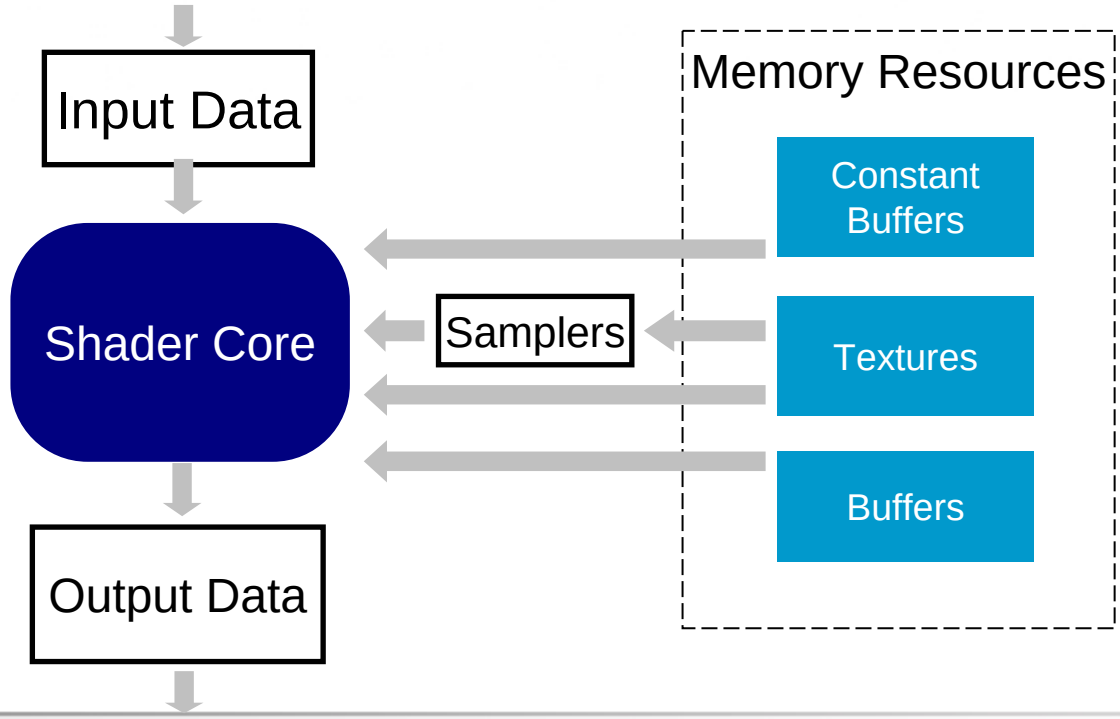
- Allows outputting VS or GS results
  - Works with variable number of output primitives
  - DrawAuto() function for unknown number of generated primitives
- All topologies are converted to lists on output
  - Might want to use points instead of triangles for intermediate passes
- Alternative to render to texture/VB

New Cool Things You Can Do

- Compute per-primitive entities
  - E.g. face normals
- Shadow volume extrusion
- Particles and particle systems
- Not really designed for high-performance tessellation, but…

Shader Model 4.0 (SM 4)

- Unified shader core for VS, GS and PS
- Only HLSL supported (no coding in assembly, but can view asm for debugging)



Shader Mode 4.0

- Load-time or offline compilation supported
- No partial precision, IEEE FP32 only
- Full (x,y,z,w) screen space coordinates can be input to pixel shader
- Textures and samplers are now independent
- Render target index supported
- 16 VS inputs
- 16 PS inputs (32 with GS)

SM4 HLSL Resource Access

- Texture functions now templated

  <T>.GetDimensions(...) <T>.Sample(...)
  <T>.SampleCmp(...) <T>.SampleCmpLevelZero(...)
  <T>.SampleGrad(...) <T>.SampleLevel(...)

- New function to load data from resources

  <T>.Load(...)

- Template types:

  Buffer
  Texture1D Texture1DArray
  Texture2D Texture2DArray
  Texture2DMS Texture2DMSArray
  Texture3D

SM4 HLSL Flow Control Management

- Dynamic Flow Control available at all shader stages
- Improved flow control management through attributes

  [branch] [flatten] [unroll(x)] [loop]

- Examples:

```
[branch] if (dot(N, L)>0) {...}
[unroll(8)] while (1) { ... }
```

- Flow control restrictions still apply!

Shader Mode 4.0 Limitations

- Practically “unlimited”

	SM 3.0	SM 4.0
Instructions	512	unlimited
FC nesting limit	4/24	32/64
Temporaries	32	4096
Indexable temporaries	0	4096
Constants	224	16x4096
Interpolators	10	16/32
Samplers	16	16
Textures	16	128
MRT outputs	4	8

Integer Shader Operations

- Supports int, uint types
- Integer and bitwise operations
  - +, -, *, /, min, max
  - not, and, or, xor, <<, >>
- Type casting and reinterpretation

```
uint uMyInt= (uint)fMyFloat;
uint uMyInt = asfloat(fMyFloat);
```

New Cool Things You Can Do

- Predictable and well defined data access in any shader type
  - Can solve problem in the best shader stage
- Complex materials
  - RenderMan movie complexity shaders on GPU
- Integer tricks
  - Custom data packing and compression

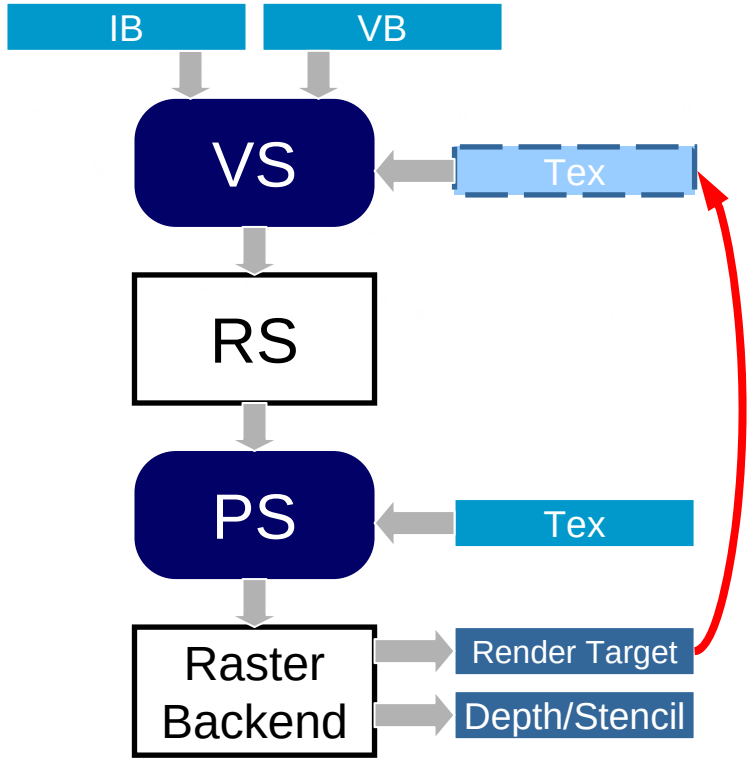
Simulating DX10 with DX9

- Can’t really simulate API
- Only some of the features can be used
- Useful to try high-level concepts
- Useful for “guesstimating” DX10 performance
- Faster than RefRast

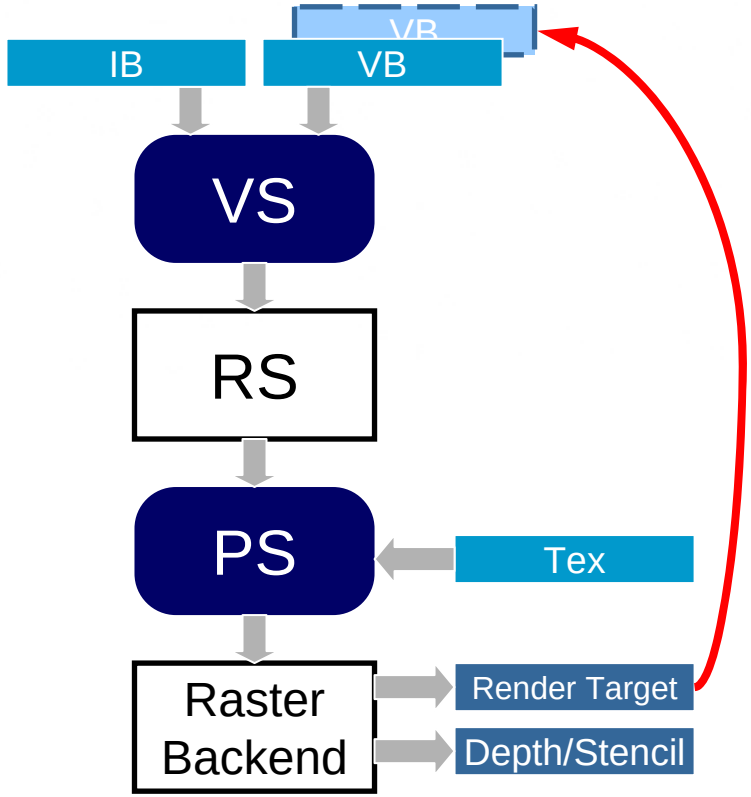
Simulating DX10 with DX9

- Idea: re-circulate data
  - Use multi-pass approach
- Can somewhat simulate GS + SO
  - Can’t generate variable number of primitives
- Features available in DX9
  - Vertex texture fetch (Nvidia)
  - Render to VB (ATI)

Multi-pass with VTF

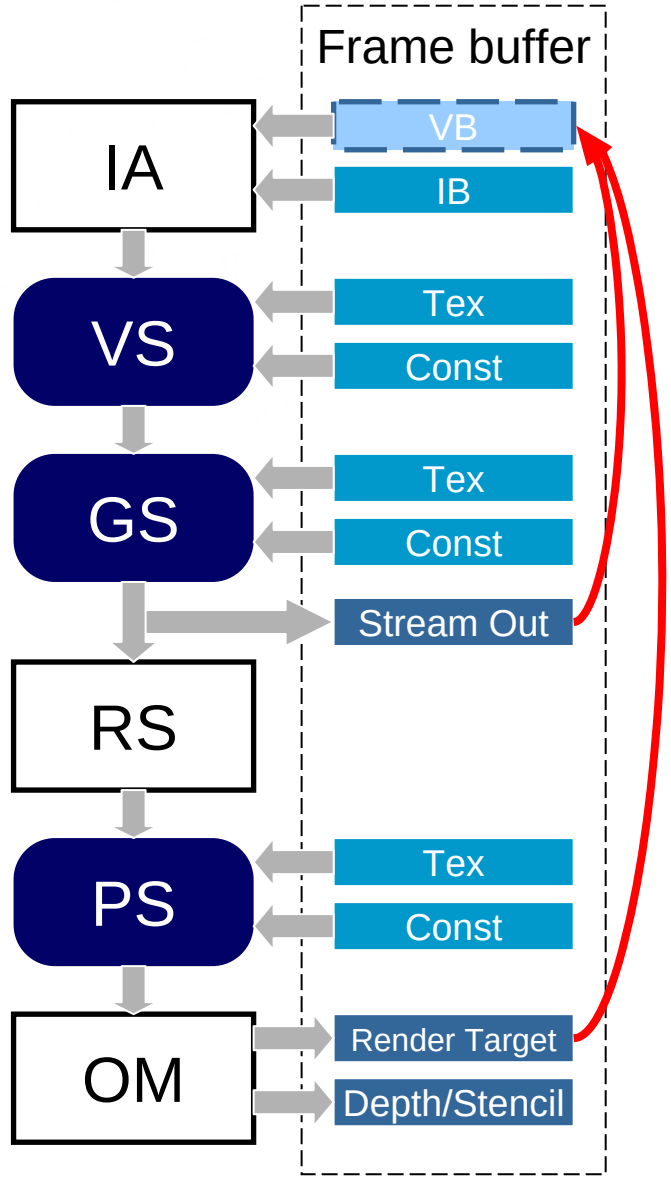


Render to VB


VB
IB
VB
VS
RS
PS
Tex
Raster
Backend
Render Target
Depth/Stencil

Render to VB in DX10

- Multiple ways to produce output
  - Stream out
  - Render targets
- Ideal rendering method in DX10 – combination of GS, stream out and render to VB
  - Need to experiment



Experimental Techniques

- Animation
- Shadow volume extrusion
- Tessellation
- Displacement mapping
- Sort on GPU
- Uber-shaders
- GPGPU

Animation

- Problems in DX9
  - Limited number of bones
    - Even worse when trying to use instancing
  - Constant uploads are expensive
  - Animation code in VS is executed on every pass

Animation

- Solution in DX10
  - Constant buffers provide large storage
  - Constant buffer updates should be cheaper
    - Could preload all animation data – no transfer
  - Stream out animated data for multi-pass
- Other possible improvements
  - Move animation blending to GPU
- Used by other techniques

Constant Buffers

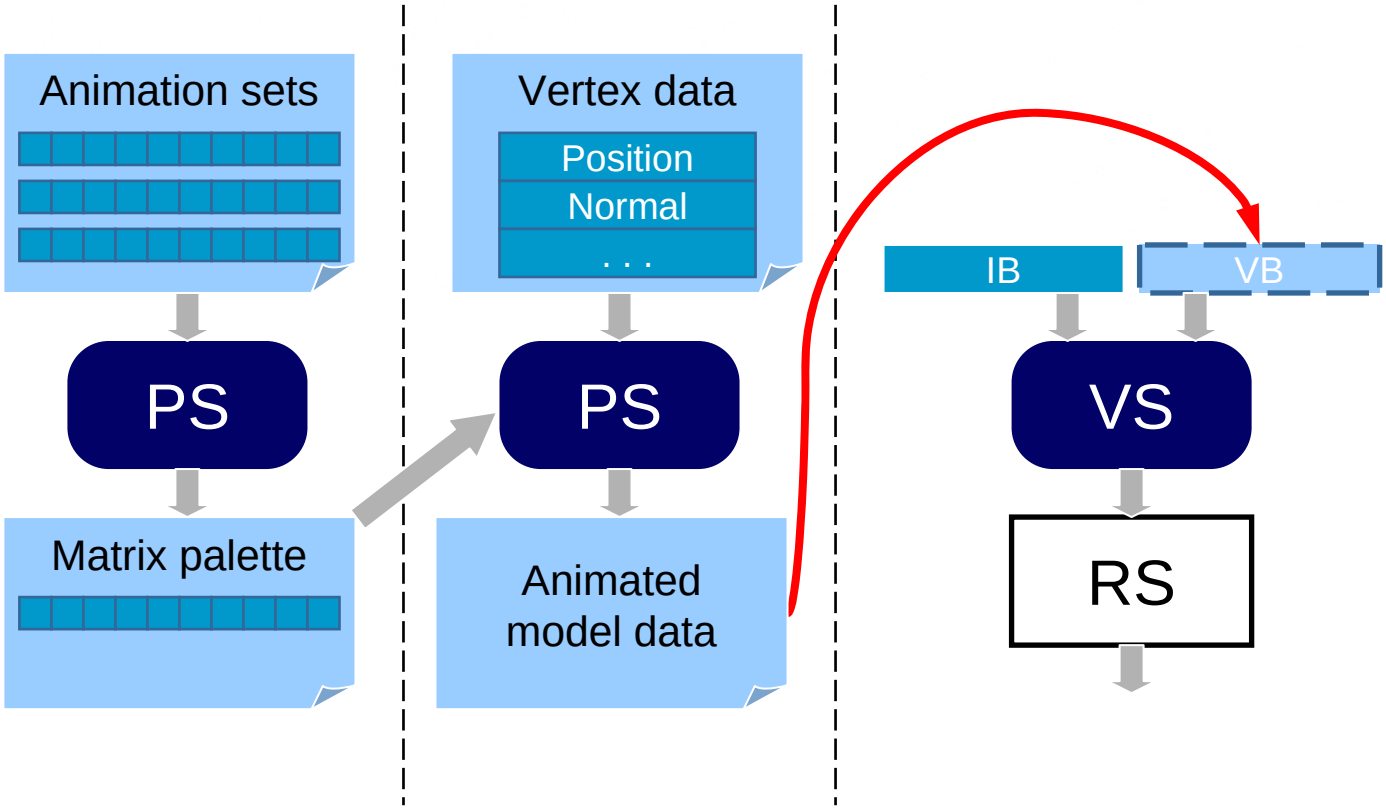
- Fake with textures in DX9 for simulation
  - Was used with render to VB animation
- DX9 VS constant update ~400 Mb/s
- DX9 dynamic texture update ~2.5 Gb/s
  - Updates batched into a large texture to reduce overhead of texture locking

Streamed Out Animation in DX10

- Pass 1
  - Draw list of points (verts) for animation and stream out
- Pass 2
  - Reinterpret animated verts with proper index information for final rendering

Streamed Out Animation in DX9

- Fake stream out with render to VB



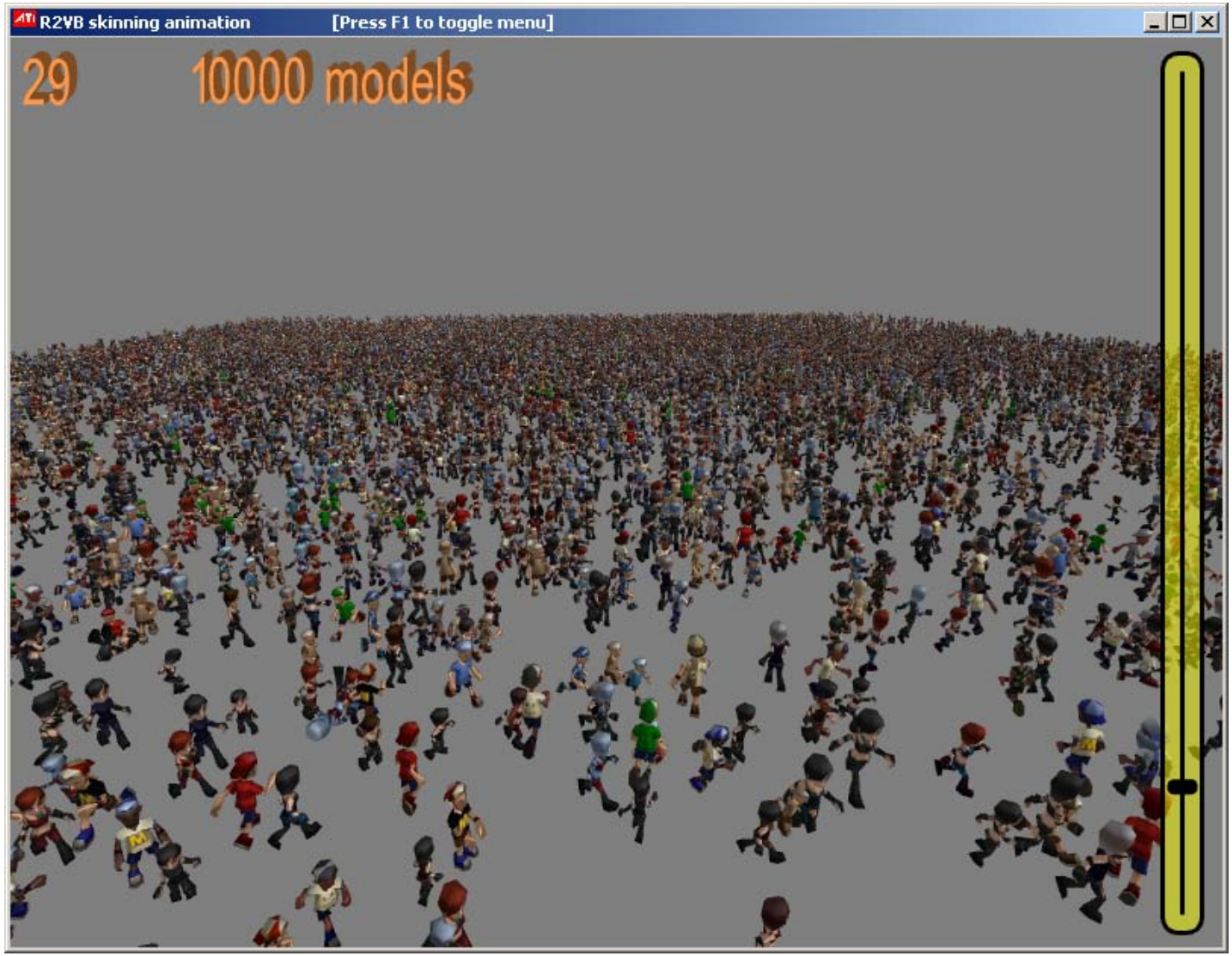
Render to VB Animation Performance

- Matrix palette generation
  - ~60-80 instructions per matrix
  - Overall negligible performance impact
- Animation in PS
  - ~80-100 instructions/vertex max depending on shader complexity
  - Mostly texture fetch bound for the bone matrices
  - 90-275 Mvert/s (depends on number of bones)

Solving Batching Problem

- Also solves batching problems
- Can batch transformations from multiple objects
- Simulate DX10 texture arrays using texture atlases
- Demo renders up to 4096 objects in one draw call

Animation Example



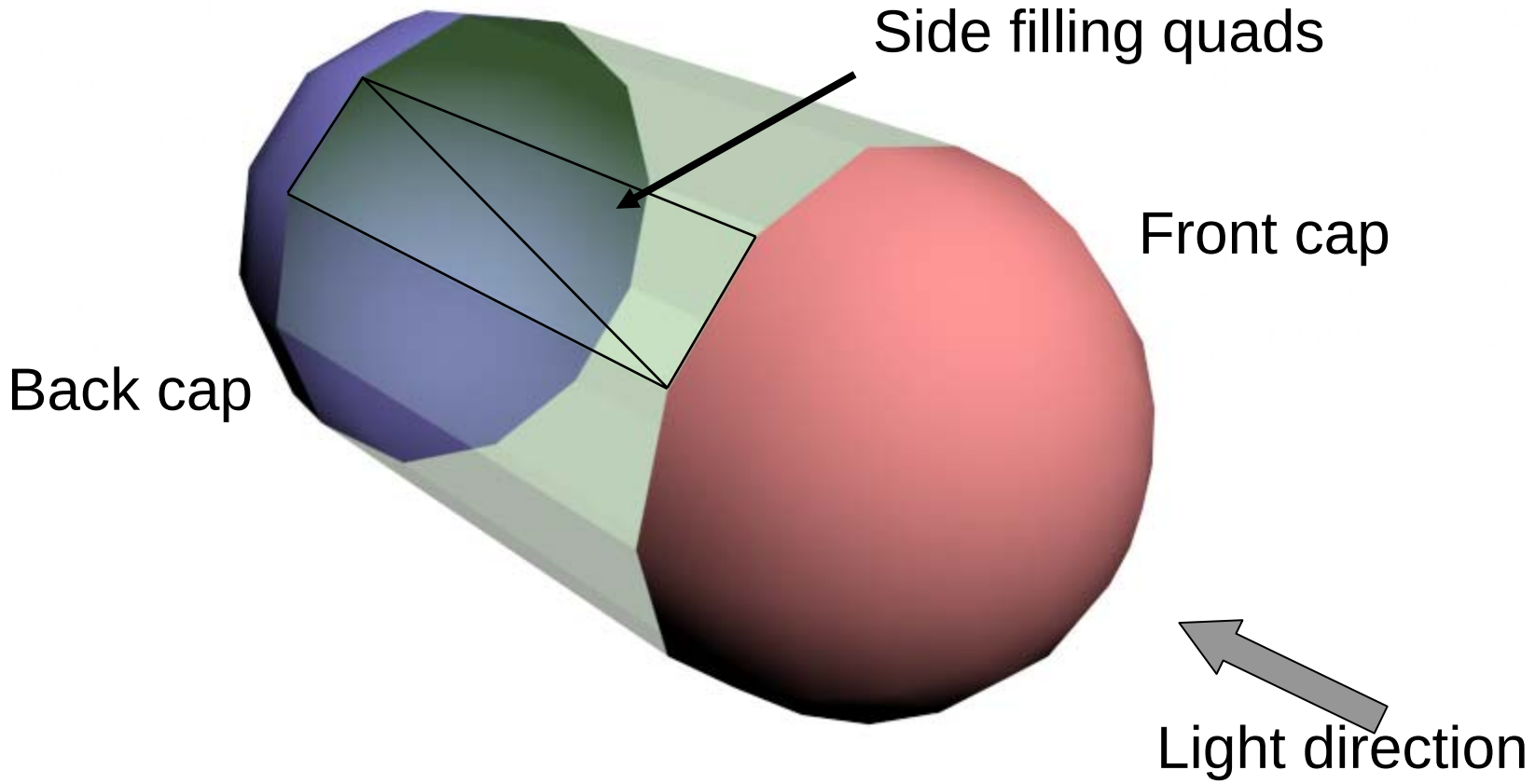
Shadow Volume Extrusion

- Problems in DX9
  - Doesn’t work correctly with animated objects
  - Games are forced to perform animation and extrusion on CPU
- Solution in DX10
  - Perform animation on GPU, stream it out
  - Perform extrusion in GS, generate sides of shadow volume

Shadow Volume Extrusion in DX10

- Compute face normal in GS
- Leave polygons facing the light in place (front cap)
- Move back-facing polygons away from light (back cap)
- Create side quads to stitch front and back caps

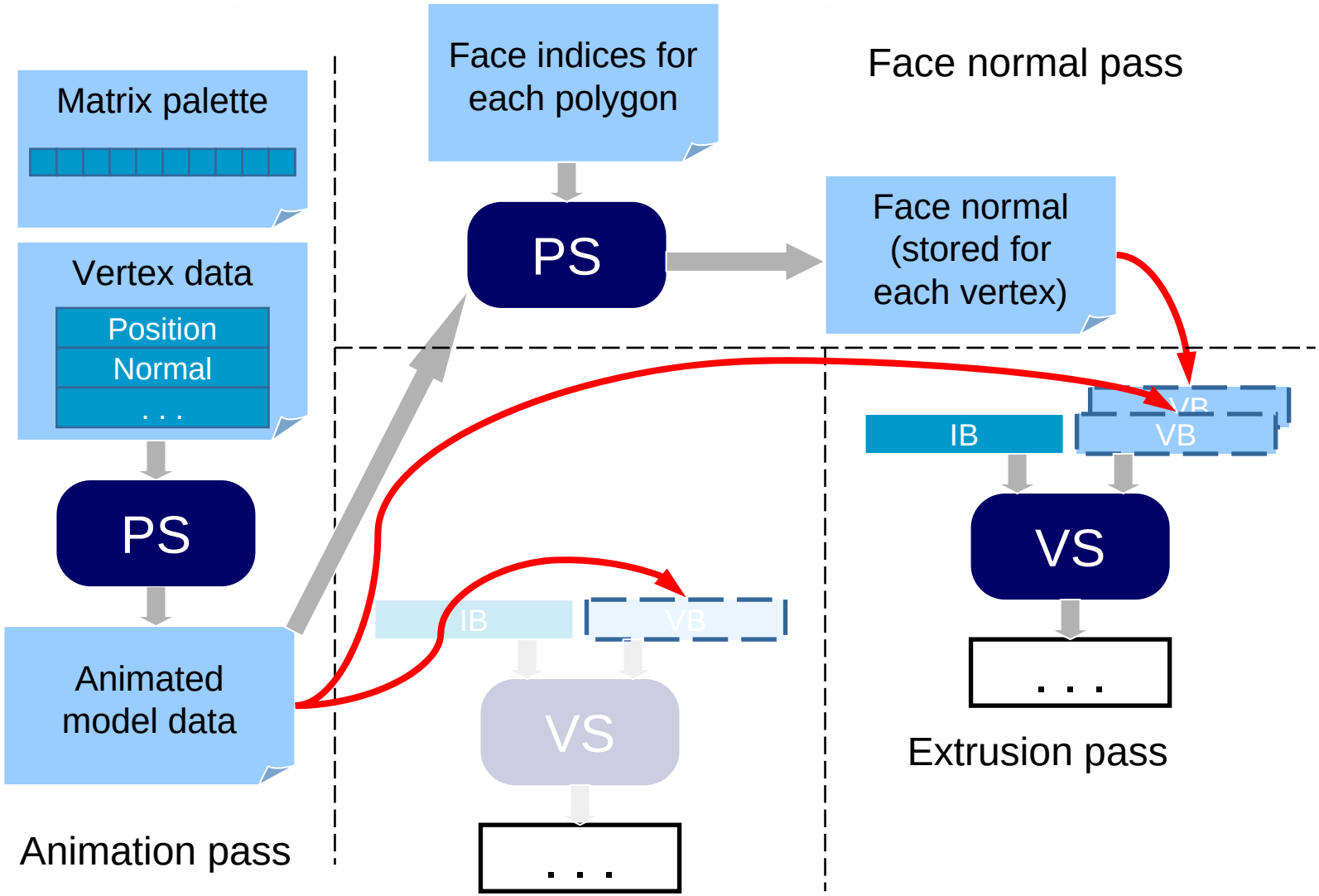
Shadow Volume Extrusion in DX10



Shadow Volume Extrusion in DX9

- Can’t generate unknown number of polygons with render to VB
- Solution: use degenerate quads for all edges
  - Some already use in DX9 for static objects
- Use separate pass to re-compute face normals after animation for the extrusion

Shadow Volume Extrusion in DX9



Shadow Volume Extrusion Performance

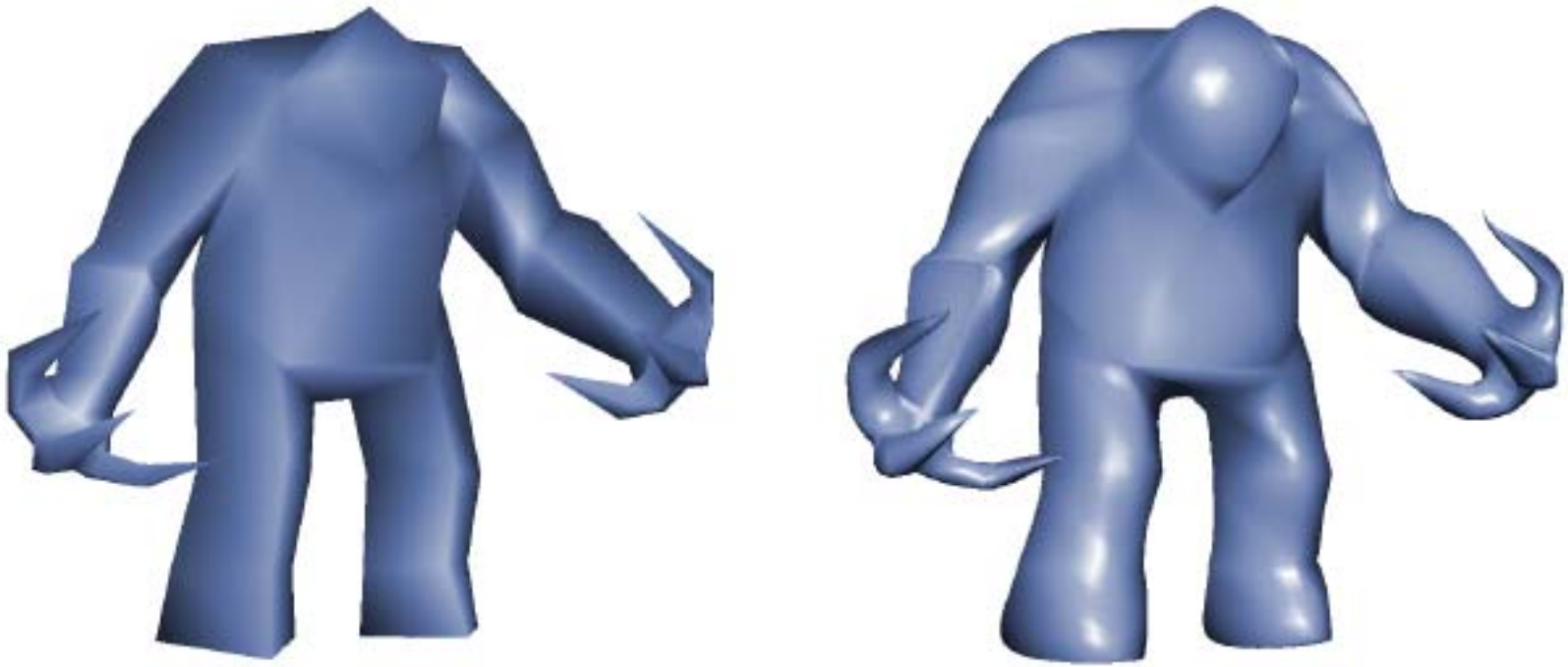
- Assumptions for extrusion in DX10 (with GS)
  - Animation – 60 instr/vertex
  - Average extrusion in GS – 40-50 instr/tri
- Assumptions for extrusion in DX9
  - Animation – 60 instr/vertex
  - Face normal calculation – 20 instr/vertex
  - Extrusion in VS – 10 instr/vertex
- Estimates: DX9 is ~20% slower than GS
  - For 1K model with moderate vertex reuse
  - Extra memory bandwidth isn’t taken into account

Tessellation

- Could use render to VB to generate fixed number of vertices/polygons
- Can simulate fixed level tessellation
- GS isn’t really designed for tessellation, but might be worth trying anyway

N-patches with Render to VB

- Back to the future ☺

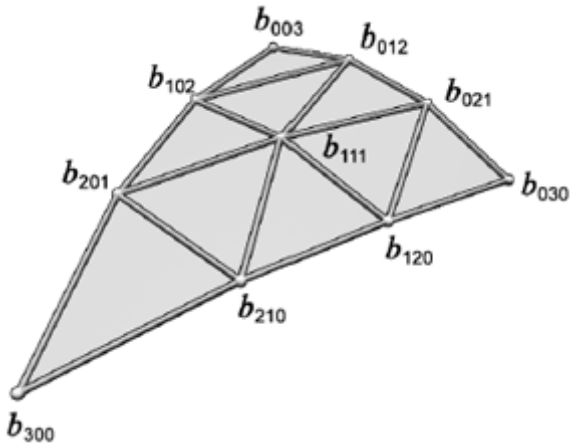


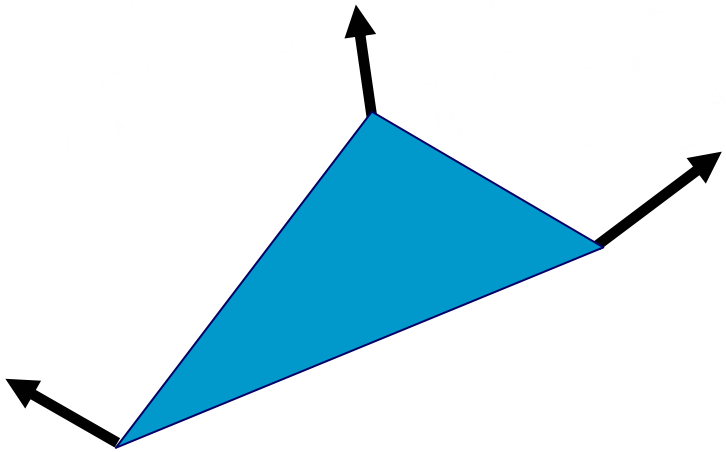
N-patches

- Cubic interpolation for the position
  - Uses triangular Bezier patch
- Quadratic interpolation for the normal
  - Different control mesh from position
- Linear interpolation for texture coordinates

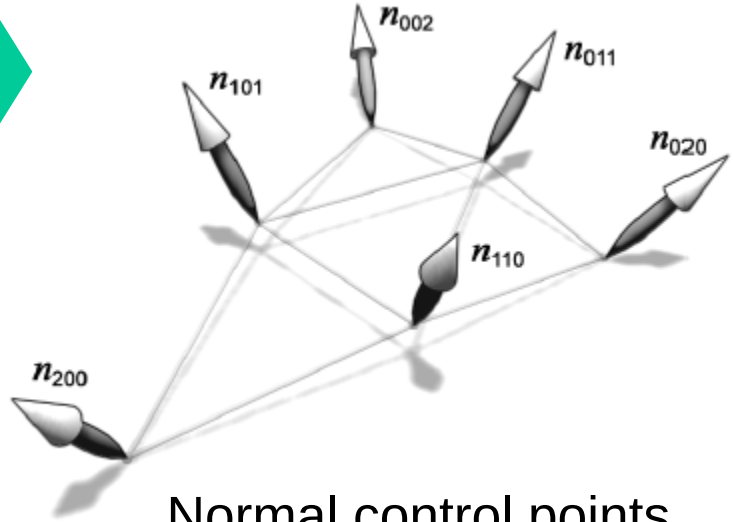
N-patches

Triangular Bezier patch





Original geometry



Normal control points

DX9 Implementation Details

- Render quad with as many pixels as final number of vertices
- For each rendered pixel:
  - Fetch 3 original control points
  - Get barycentric coordinate of this vertex
  - Compute new position, normal and tex coord
- Use pre-computed barycentric coordinates to compute new positions, normals, etc.
  - Store in (N x 1) texture, N – verts per patch
- Finally render new mesh

N-patch Performance

- DX9 performance
  - All control points are re-computed for every vertex – a bit wasteful
  - Tessellation shader – 112 SM 3.0 instructions
  - Tessellation rate – 60-80 Mverts/sec
- DX10 improvements
  - GS removes redundant control point computations, but other bottlenecks could show up

Displacement Mapping

- Displace vertices with VTF
- Could use with or without tessellation
- Usage examples
  - Geometry compression technique
  - Procedural or dynamic displacement
  - Creation of unique objects
  - Intersection / collision detection

Geometry Compression

- Can create huge landscapes
  - Use really small footprint formats
- E.g. can store heightmap of California in 222Mb with 30 x 30 m resolution using BC4_UNORM format
  - 24:1 compression ratio

Dynamic Displacement

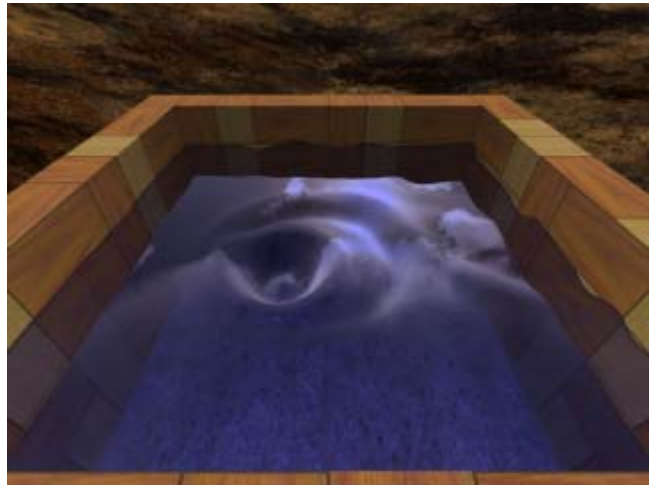
- Compute displacement/heightmap
- Re-compute normals in texture space
  - Use Sobel filter (could be done with 4 bilinear fetches)

-1	0	1
-2	0	2
-1	0	1

Filter for dX

-1	-2	-1
0	0	0
1	2	1

Filter for dY



Dynamic Displacement

Snow Example

- “Physically correct” snow accumulation
- Making imprints in texture space
  - Render parts of objects intersecting snow
  - Blur imprints to create more natural slope
  - Combine new imprints with previous layer of snow
- Varying depth of imprints based on snow depth

Automatic Object Placement

- Automatically put objects (trees, grass, etc.) on the terrain
  - Use displacement map to find vertical displacement
- Works with procedurally generated landscapes
- Works perfectly for objects where CPU doesn’t need to know real position
  - Just keep that data on GPU

Creating Unique Objects

- Make all your objects look different using the same mesh
- Displace basic mesh using collection of displacement maps
  - E.g. no 2 trees in the forest are the same

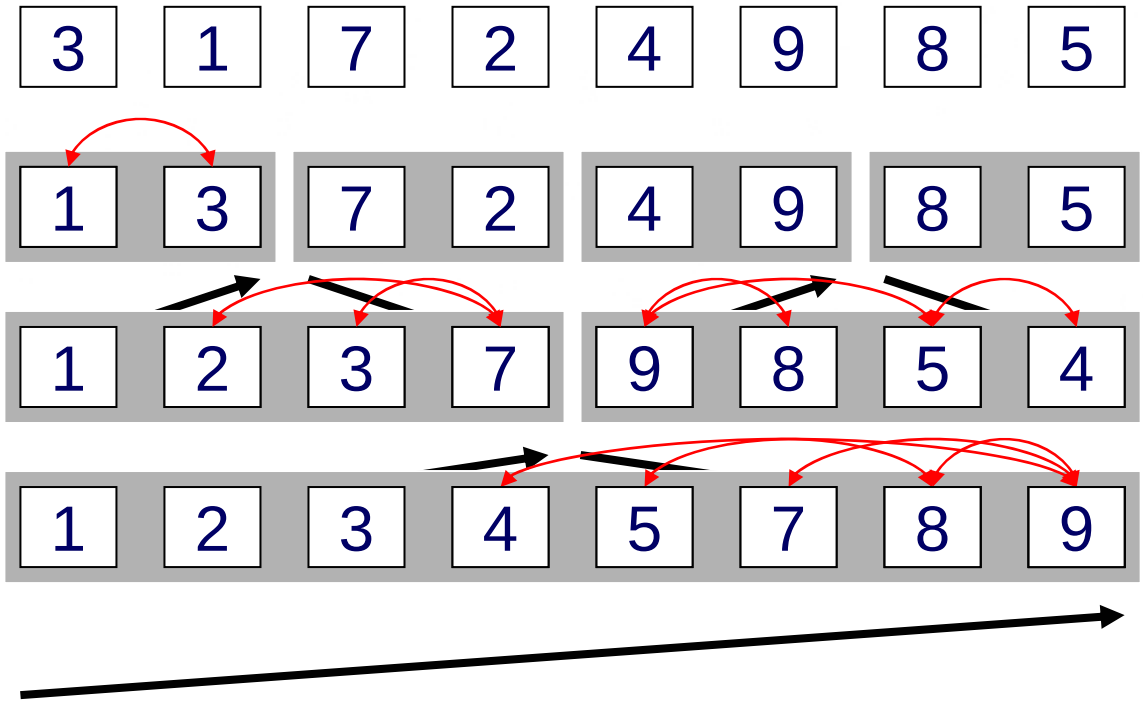
Sorting on GPU

- Order independent transparency is a huge problem
- Could sort alpha blended particles (or other polygons, objects, etc.) on GPU
- Works perfectly with particle systems implemented on GPU

Bitonic Sort

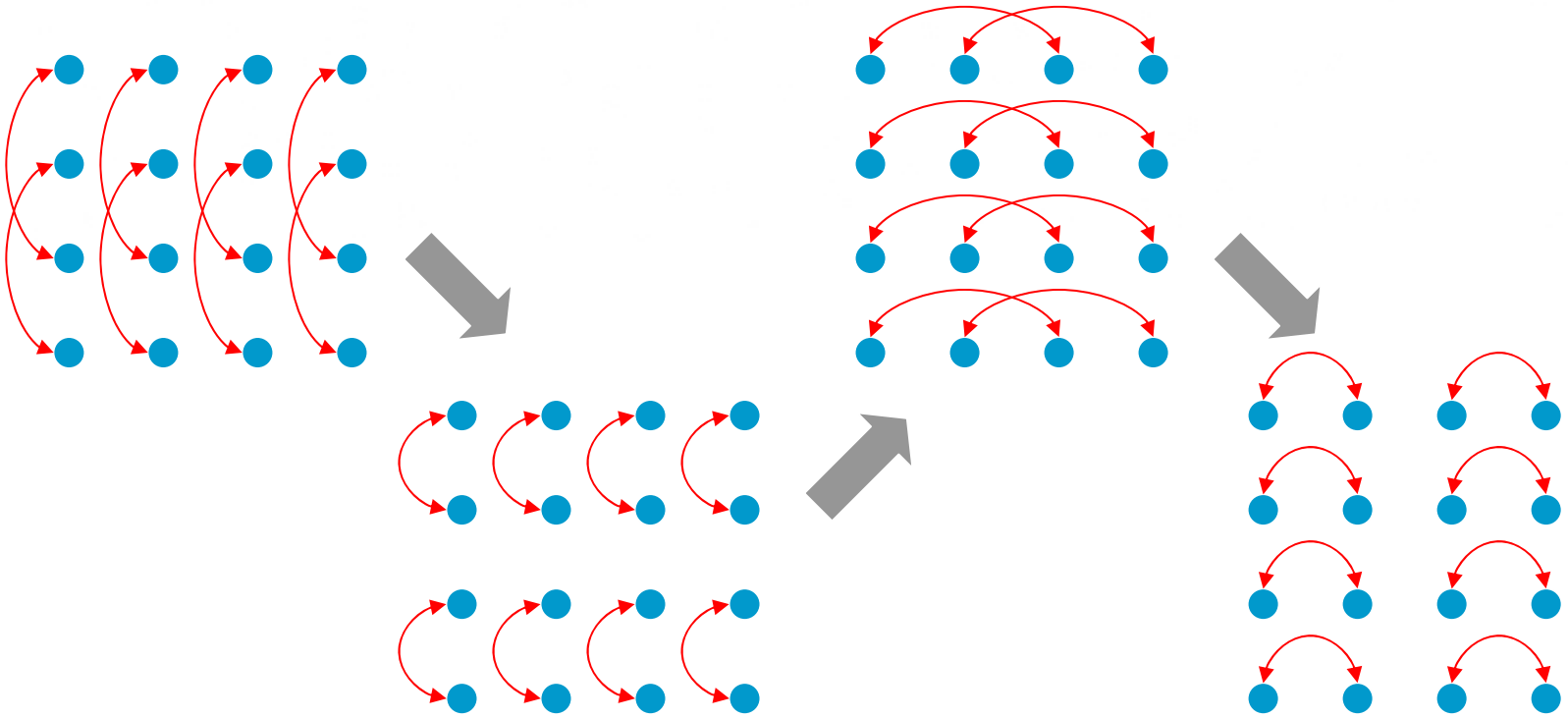
- Very hardware friendly and has excellent parallelism
  - Can be implemented without recursion
- Performance: $\Theta(n\ (log_2\ n)^2)$
  - 16K particles at over 40 fps on X1900
- 2D texture used to store and sort more than 4096 particles

Bitonic Sort Example



2D Bitonic Sort

- Combination of vertical and horizontal sort passes



Last 4 sort passes

Implementation Tricks – Part 1

- Use 2 textures due to number of data components
  - Position, distance, particle ID
- Compare-and-exchange operation
  - Twice the workload
  - Example:
    - Compare A and B, smaller goes into A
    - ... then B and A, greater goes into B

Implementation Tricks – Part 2

- Use texture coordinate “magic” to select direction of pixel offset
- Use square wave with +1 and -1 values to drive direction of the offset and sign of comparison

Bitonic Sort Shader (VS)

```
struct VsOut {
        float4 pos: POSITION;
        float4 texCoord: TEXCOORD0;
        float3 dir: TEXCOORD1;
};

float2 halfPixel, select;
float step0, step1, step2;

VsOut main(float4 pos: POSITION)
{
        VsOut Out;
        Out.pos = pos;
        Out.texCoord.xy = pos.xy * float2(0.5,-0.5) + 0.5 + halfPixel;
        Out.texCoord.zw = 0;
        // Select sample direction
        float r = dot(Out.texCoord, select);
        // Sample direction and frequencies
        Out.dir = float3(r*step0, r*step1, Out.texCoord.y*step2);
        return Out;
}
```

Bitonic Sort Shader (PS)

```
struct PsOut {
        float4 pos:  COLOR0; // particle position
        float4 dist: COLOR1; // distance used for sorting
};

sampler2D Tex;
sampler2D Dist;

// Distance between pixels to compare.
float4 offset;

PsOut main(float4 texCoord: TEXCOORD0, float3 direction: TEXCOORD1){
        PsOut Out;
        // Sample current value
        float4 d0 = tex2D(Dist, texCoord.xy);
        // dir.x = Sample and comparison direction.
        // dir.y = Horizontal comparison frequency.
        // dir.z = Vertical comparison frequency.
        float3 dir = (frac(direction) < 0.5)? 1 : -1;
        // Select other value to compare to
        float4 texCoord1 = texCoord + dir.x * offset;
        float4 d1 = tex2D(Dist, texCoord1.xy);
. . .
```

Bitonic Sort Shader (PS)

```
. . .
        // Compute comparison direction and make actual comparison
        dir.x *= dir.y;
        dir.x *= dir.z;
        dir.x *= (d1.x - d0.x);

        // The sign indicates which value we want
        if (dir.x <= 0){
                Out.pos = tex2D(Tex, texCoord);
                Out.dist = d0;
        } else {
                Out.pos = tex2Dlod(Tex, texCoord1);
                Out.dist = d1;
        }
        return Out;
}
```

Mixing Materials

- Often alpha blended particles have different textures or even materials
- In DX10 use texture arrays and uber-shaders to render all in one pass
  - Flow control in uber-shader will have good coherency
- GS can even generate different geometry for different types of particles

Uber-shader Example

- Grass – static particles with different textures
- Fires – vertically rotating billboards with procedural fire (generated in PS)
- Smoke – camera oriented particles

Sorting Demo

GPGPU

- Physics on GPU allows keeping data on the card all the time
  - No extra data transfers
  - Free up CPU for more important tasks
- The best solution for GPU generated particle systems, etc.
  - E.g. smoke interaction with the ground

Conclusions

- DirectX 10 overview
- New pipeline and features
- Effect ideas and DirectX9 fallbacks

Questions