CS195V Week 6

Image Samplers and Atomic Operations

Administrata

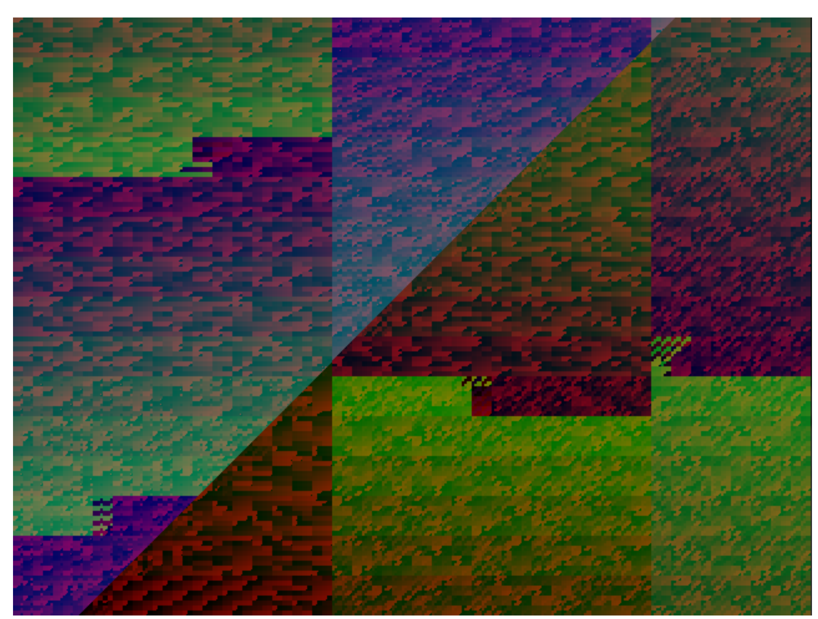
- Warp is due today!
- NBody should go out soon
  - Due in three weeks instead of two
  - Slightly larger in scope
  - First week and a half we be spent on implementing a cs32 Solar-like sim
  - Second week and a half will extend that to fluid simulation
- We need more case studies!

Texture Buffer Objects (TBOs)

- Probably won't use them (at least not yet), but worth knowing what they are
- Introduced in OpenGL3.x
- 1 dimensional storage as a texture
- Cannot do any filtering and must be accessed using texelFetch (ie. coordinates must be integer index, not [0, 1])
- Useful if you want to pass large uniform arrays to a shader

OpenGL 4.2

With OpenGL 4.2 you can now do this:



(╯°□°）╯︵ ┻━┻)

OpenGL 4.2

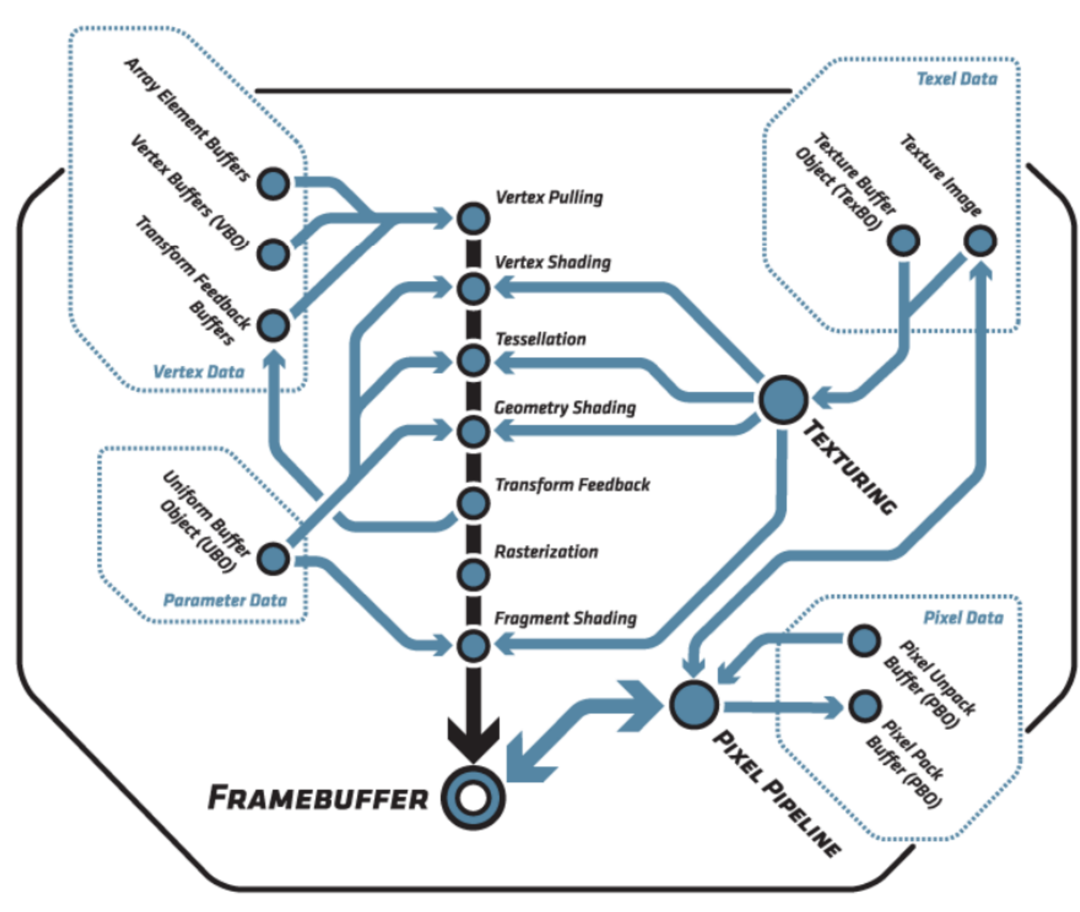
- OpenGL 4.2 official at least year's SIGGRAPH
- Up until now we focused on OpenGL 4.0 specific features
- Keep in mind that the drivers for the GPUs in the CS department only have 4.1 support
  - How to check: `glxinfo | grep "OpenGL version"`
- This means that some of the functionality / naming conventions may not follow the 4.2 specifications exactly (ie. don't trust the docs or the language spec)

Using glxinfo

- glxinfo is your friend
- running glxinfo will spit out all the data you ever need about gfx support
    - Probably want to grep stuff since this is a lot
- Example : discovering which extensions are supported
    - glxinfo | grep GL_NV_
    - glxinfo | grep GL_EXT_
- Example : determining your video card
    - glxinfo | grep string

The Current OpenGL Pipeline

No longer really a pipeline, more like the NY subway system



OpenGL 4.2

- Because we're on 4.1 drivers, this means we need to declare extensions when we want to use 4.2 functionality
- In GL, you should append EXT to 4.2 functions
- In GLSL, you need to add

#extension GL_NV_gpushader5 : enable

- Plus whatever extension defines you plan to use (different for each 4.2 feature)

OpenGL 4.2 New Features

- Shader Atomic Counters
  - Atomic functions to increment / decrement counters
- Shader Image Load Stores
  - Atomic functions to read / write from a single texture level
- Texture Storage
  - Immutable textures
- Transform Feedback Instanced
  - Draw multiple instances of a transform feedback object
- Shading Language 4.2 pack
  - New GLSL features

OpenGL 4.2 New Features

- Texture Compression BPTC
  - Support for new compression format (same format as BC6H / BC7 in DX11)
- Base Instance
  - Specify offset within buffer objects for instanced drawing
- Internal format query
  - Query supported samples for an internal format
- Plus many performance improvements
  - Arbitrary modification of a subset of a compressed texture (requested by Blizzard)
  - Small data type packing into larger, GLSL shaders can r/w to 16 bit floating point encodings
  - and more....

Immutable Texture Storage

- Texture structure is immutable on creation (format, dimensions, mipmaps)
- Contents are mutable

```
void TexStorage{1D,2D,3D}(enum target, sizei
levels, enum internalformat, sizei width,
sizei height, sizei depth)
```

- Addresses OpenGL's rather ad hoc resource management (the drivers don't know anything about the texture at the beginning of a draw...)
  - ie. might need to alloc more memory for mipmaps, resulting in poor allocation patterns and performance losses

Shader Atomic Counters

- Probably won't use much, although this is a pretty important extension
  - one of the reason's we're not using it is I don't think the cards even support this...
- Allows for a GPU global atomic counter
  - Atomic means that they are single unbroken operations
  - No critical problems across multiple shader invocations, but some ambiguity, as we will see later

Shader Atomic Counters

http://www.opengl.org/registry/specs/ARB/shader_atomic_counters.txt

- Defines a new type, `atomic_uint`
- Useful functions:

  uint atomicCounterIncrement(atomic_uint)

  Increments & returns val prior to increment

  uint atomicCounterDecrement(atomic_uint)

  Decrements & returns val after decrement

  uint atomicCounter(atomic_uint)

  Returns val
- Note that shader atomic counters are much faster than image atomic functions on AMD hardware, although this might not be the case for NVIDIA hardware

Shader Image Load Stores

- These are probably the most exciting part of GL 4.2
- Shader image load stores allow shaders to load, store, and perform atomic read-modify-write operations to an image from any shader stage
- To use this functionality in your shader, you must define

```
#extension GL_EXT_shader_image_load_store : enable
```

OpenGL Images

- An image is basically a single level of a texture (note: I think this is a terrible name...)
- OpenGL now allows you to bind a single level of a texture to an "image unit"
- You give up some things like mipmapping and automatic filtering
- In return, you have the ability to perform arbitrary read / write / atomic operations on texels

Atomic Image Operations

- You can load and store from arbitrary points in an image unit
- You can also perform operations like imageAtomic* with an immediate value
    - Max, And, Or, Swap, etc.
- Can edit textures inside the shader at any stage

Image Functions (C++)

- glBindImageTexture
    - You will have to call glBindImageTextureEXT
- Parameters: (uint index, uint texture, int level, boolean layered, int layer, enum access, int format)
    - index - the index of the image unit you want to bind to (starts at 0)
    - texture - the id of the texture you want to bind to the image unit (generated by glGenTextures)
    - level - the level of the texture to bind to (0 for full resolution)
    - layered - if you are binding a layer of a 2D texture to a 1D texture, or a layer of a 3D texture to a 2D texture
    - layer - the number of the layer to bind if you are using a layered texture, otherwise ignored
    - access - read only, write only, readwrite
    - format - format of the pixels in the image

Image Functions (shader)

- imageLoad, imageStore
- imageAtomic*
  - Does the computation, stores the result in the image, and returns the value
- Note that you index into an image using integer offsets, not normalized texture coordinates!
- You need to pass in an ivec, might require some conversion of texture coordinates and such

A Trivial Example (with color!)

```
#version 410 core
#extension GL_NV_gpu_shader5 : enable
#extension GL_EXT_shader_image_load_store : enable
coherent restrict uniform layout(size4x32) image2D image;
#ifdef _VERTEX_
    ...
#endif
#ifdef _FRAGMENT_
out vec4 out_Color;
void main() {
    ivec2 coord = ivec2(gl_FragCoord.xy);
    out_Color = imageLoad(image, coord);
}
#endif
```

Image layout qualifiers

```
coherent uniform layout(size4x32) image2D myImage;
volatile uniform layout(size1x8) uimage1D uIntArray;
coherent restrict uniform layout(size4x32) image3D APM;
```

- The layout qualifiers correspond to the texture format

```
size1x8 : R8I, R8UI
size1x16 : R16I, R16UI
size1x32 : R32F, R32I, R32UI
size2x32 : RG32F, RG32I, RG32UI
size4x32 : RGBA32F, RGBA32I, RGBA32UI
```

- Note that this is different from what is defined in the 4.2 lang spec (which will not work)

coherent

Memory accesses are done with similar accesses from other shader threads.

When reading a variable declared as "coherent", the values returned will reflect the results of previously completed writes by other shader threads.

Note that Shader memory reads and writes complete in a largely undefined order (see memory barriers).

You'll be using this most of the time

volatile

Volatile implies the image could be read/written at any point during shader execution by some source other than executing thread.

When reading a volatile, its value must re-fetched from the underlying memory, even if the thread performing read had already fetched its value from the same memory once. When writing, its value must be written to the memory, even if the compiler can conclusively determine that its will be overwritten by a subsequent write.

Since the external reading or writing may be another shader thread, volatiles are automatically treated as coherent.

restrict

Restrict variables may be compiled assuming that the variable used to perform memory access is the only way to access the underlying memory using the shader stage in question.

This allows the compiler to coalesce or loads and stores using "restrict"-qualified image variables in ways wouldn't be permitted for image variables not so qualified, because compiler can assume that the underlying image won't be read or written other code.

Incorrectly declaring a variable restrict will have undefined results.

const

Memory accesses to image variables declared using the "const" qualifier may only read the underlying memory, which is treated read-only.
It is an error to pass an image variable qualified "const" to imageStore() or imageAtomic*().

Also I'm pretty sure const doesn't work on the department machines.

readonly / writeonly

Image variables can be declared as read only and write only, and should match the call to glBindImageTexture.

However, these qualifiers seem not to be supported by the department machines.

Other...

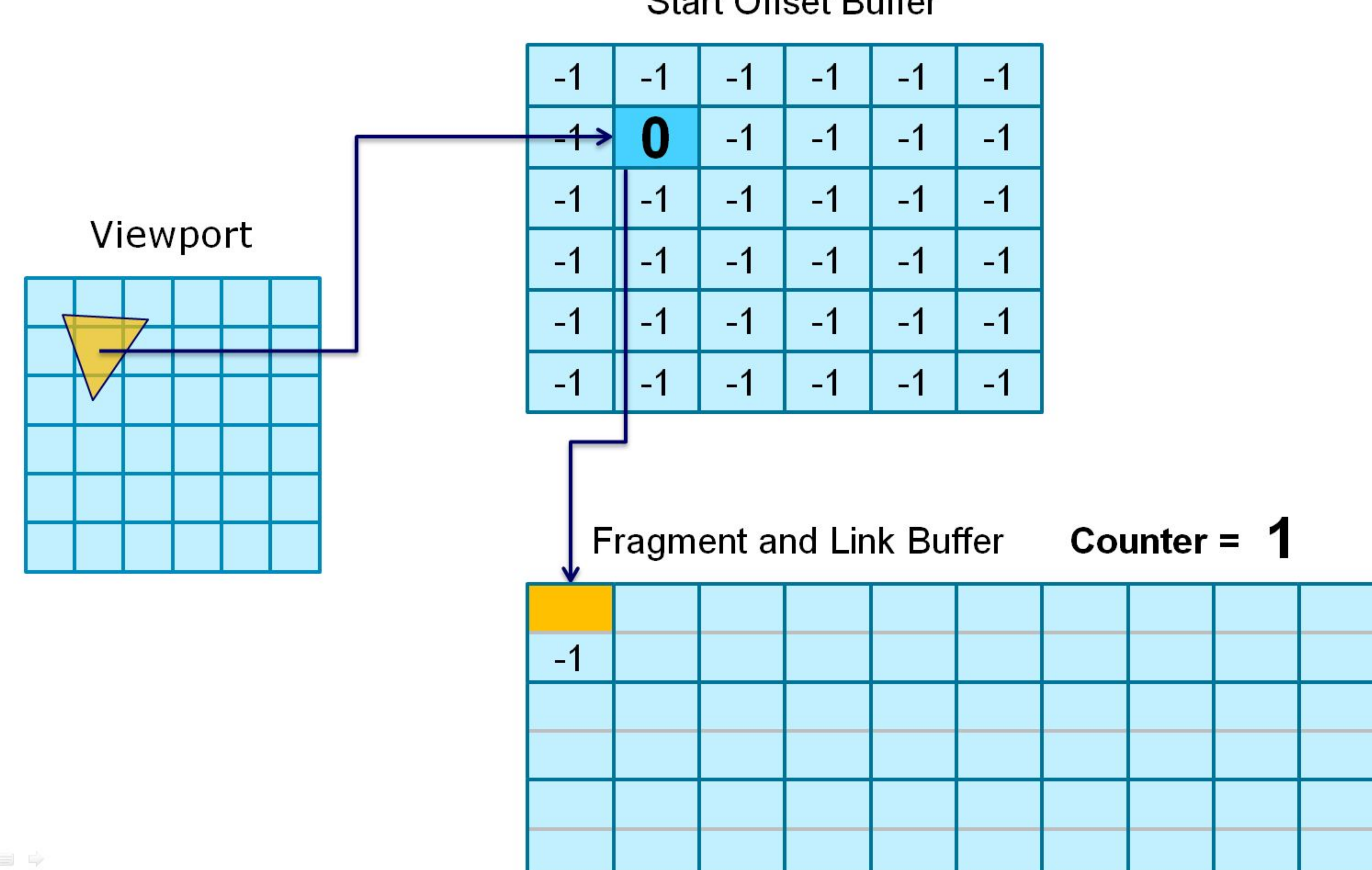
- You can also use layout qualifiers when declaring function arguments. However, these qualifiers can only add qualifications; they cannot remove them

Memory Barriers

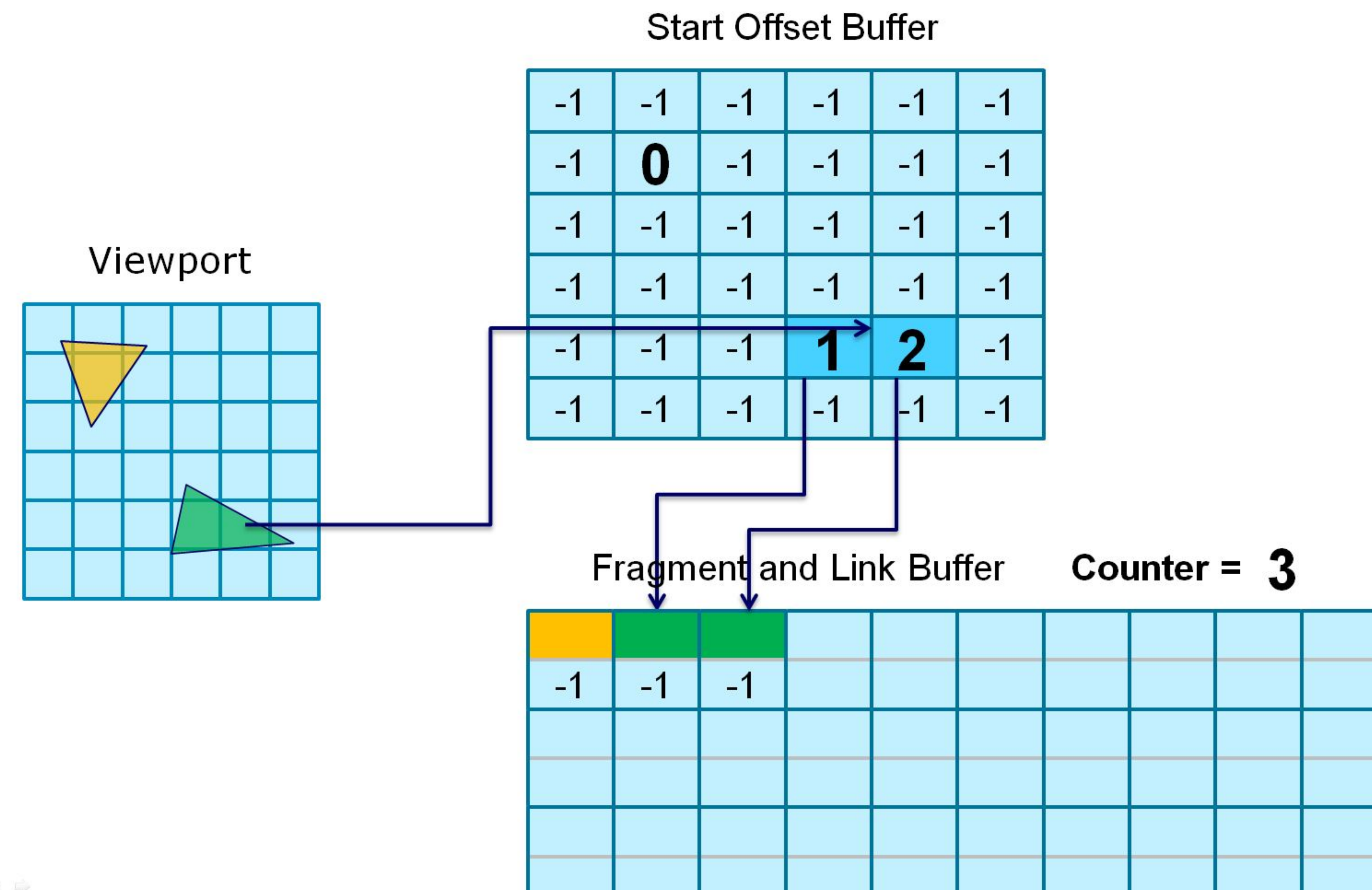
- With all of this editing going on, how do you guarantee the order of operations?
- Memory barriers (both in the shader and C++) can guarantee that all operations of a particular type finish before continuing
  - The shader version only checks for memory accesses in general, C++ can specify type of operations to wait for
- This way, you can synchronize your shaders to behave in a more predictable manner

Why Use Memory Barriers?

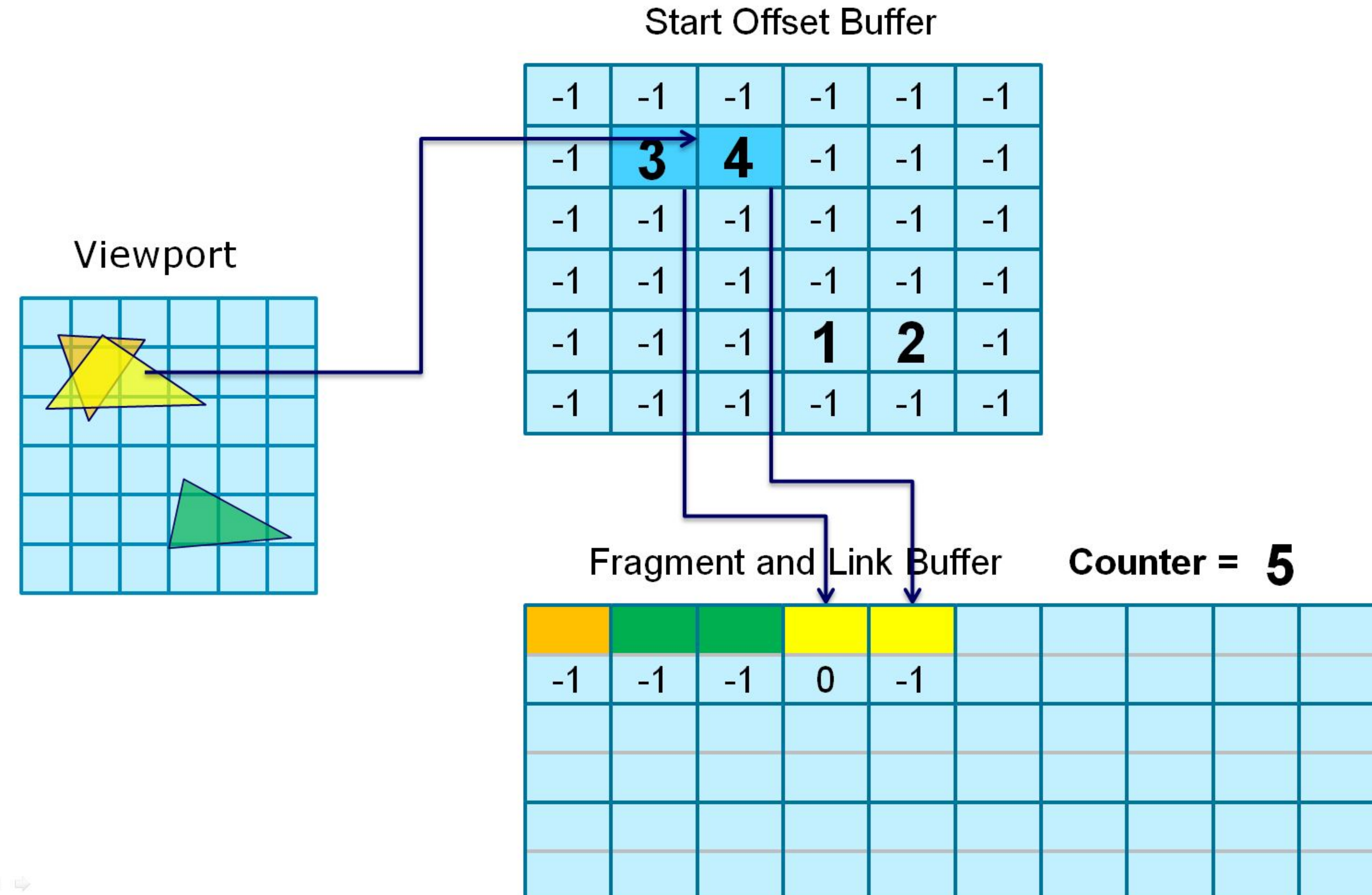
- Most of the time you don't really notice these memory synchronicity issues
  - However, with atomic operations especially, we can have problems
- For example, say you draw two primitives with the same shader (A first, B second)
  - Shader invocations on A are not necessarily guaranteed to happen before invocations on B
  - You might say "wait, but then why do later primitives always get drawn on top when we disable depth test?"
  - B will always be drawn to the framebuffer later, but actual shader operations (like image stores) are not guaranteed to happen later
- You can use memory barriers to guarantee this ordering



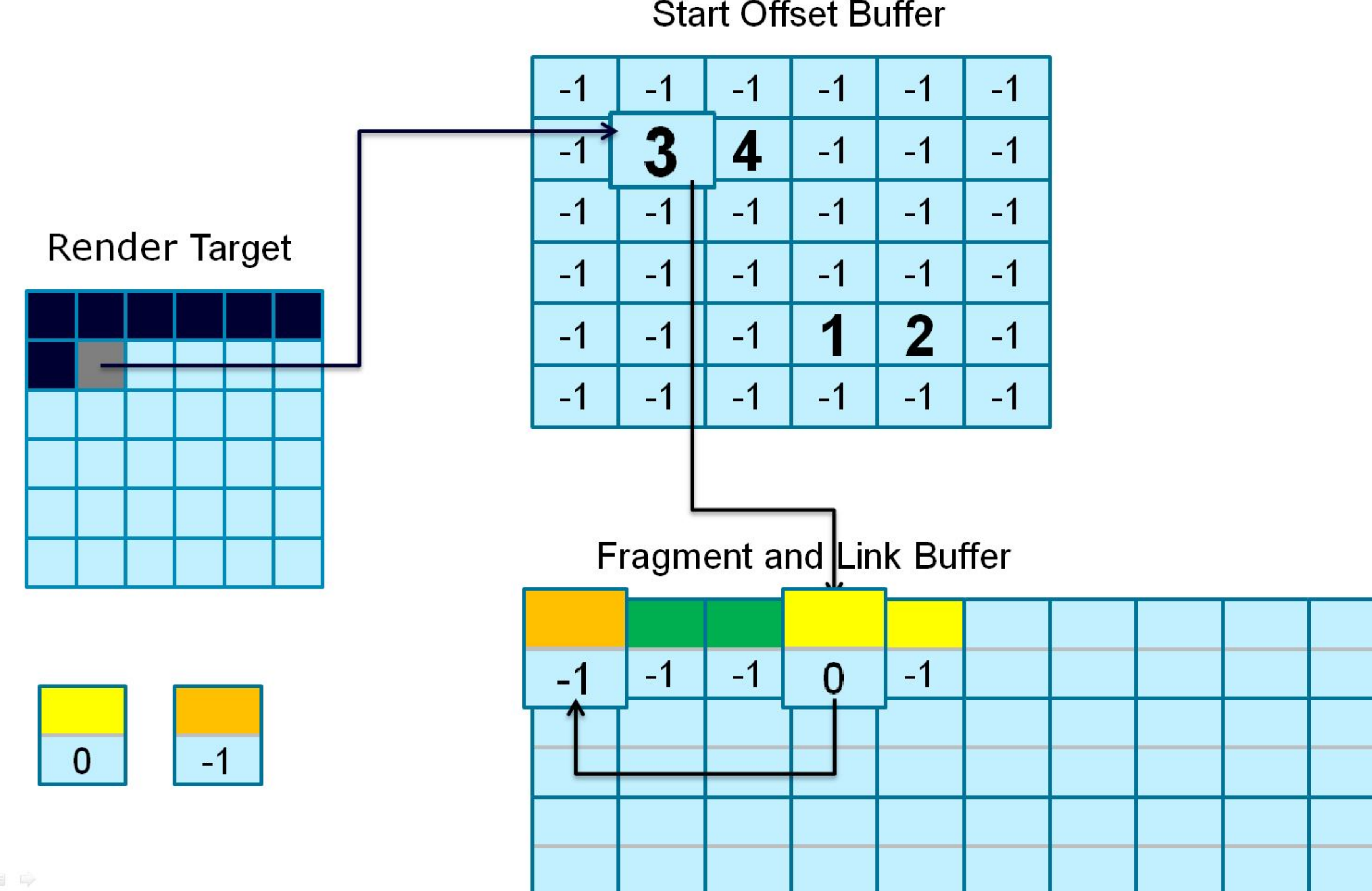
Insert



Two more inserts



Two inserts (one link push)



List access

Mini Case Study: Order Independent Transparency (AMD)

- Draw all transparent objects into the pixel list
- For each pixel, sort its fragment list and blend them together to make the final image
- Use whatever blend function you want to get the final blended transparent pixel value
- GDC 2010: OIT and GI using DX11 linked lists (Nick Thibieroz & Holger Grün)

Result

NVIDIA Grid Hashing

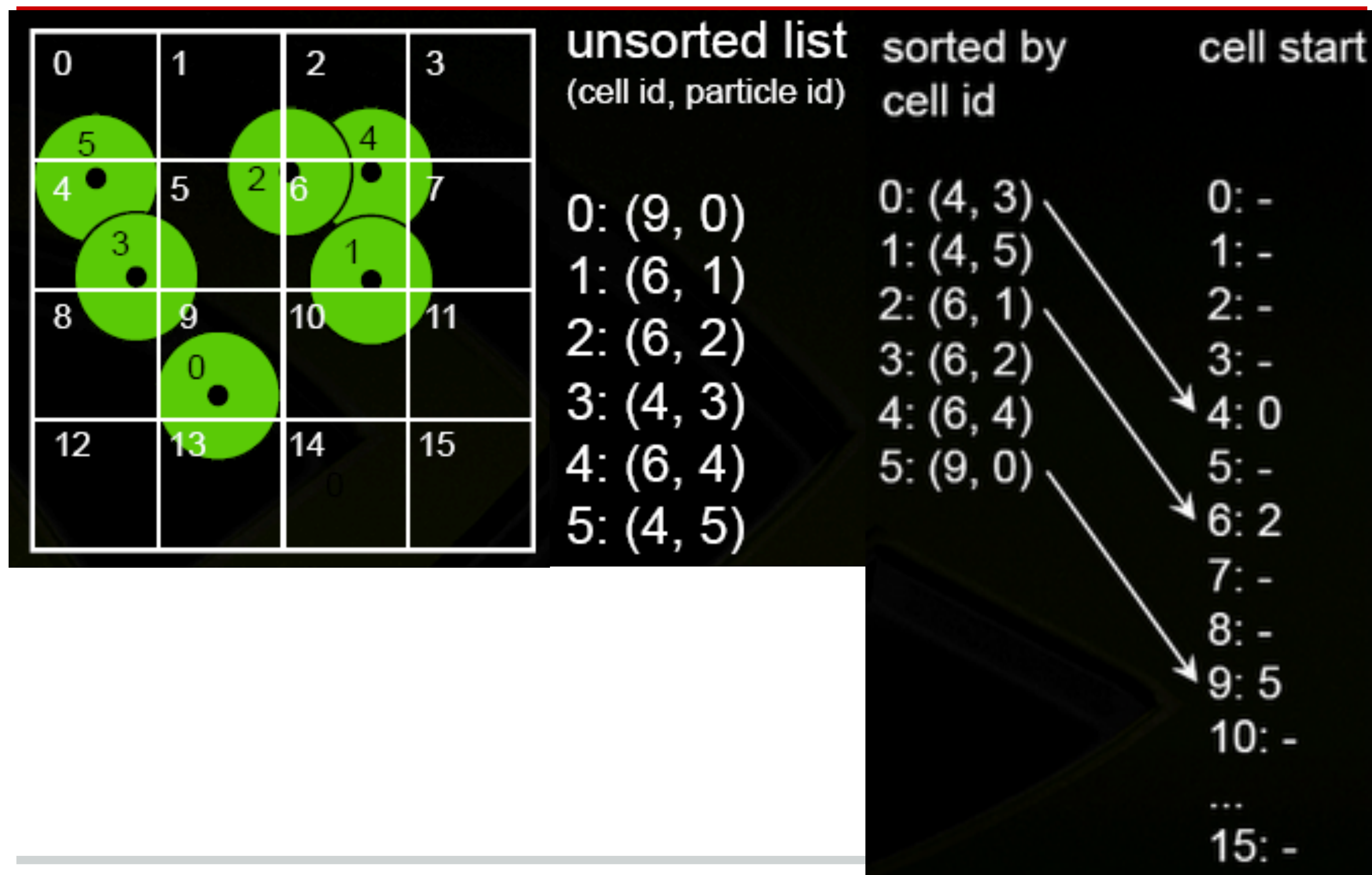
Mini Case Study

http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf

NVIDIA Grid Hashing

- Grid built from scratch each frame
- Algorithm
  - Computer grid cell for each particle (based on center)
  - Calculate cell index
  - Sort particles based on cell index
  - Store start of each bucket in a sorted array
  - Process collisions by looking at a localized neighborhood of cells for each particle

Pictures!



Depth Buffers

- You probably already know of the depth/z buffer
- Useful for a lot of things aside from depth testing
- Quick way to get some basic 3D image space information

Depth Buffer Precision

- Depth buffer values are stored as integers

z_buffer_value = (1<<N) * (a + b / z)

Where:
N = number of bits of Z precision
a = zFar / (zFar - zNear)
b = zFar * zNear / (zNear - zFar)
z = distance from the eye to the object

- This means that depth buffer precision is proportional to 1/depth, so we have more precision closer to the near plane\
- If you have depth precision problems, increase the bits in the depth buffer or move the near plane farther from the eye

http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html

Using the Depth Buffer in Your Code

- Support code framebuffer object has a depth() method which returns the texture id of the depth texture for that framebuffer
- You can bind this texture as a sampler2D just like any other texture

Basic Shadow Mapping

Case Study

Shadow Mapping with Depth Buffer

- Old, basic technique for making shadows
- Some problems with aliasing due to being an image space technique; quality is dependent on texture resolution and view
- However, a simple and effective way of generating shadows

Shadow Mapping with Depth Buffer

- Draw the scene from the light's point of view; save the depth buffer
- Next, draw the scene from the normal camera view
  - For each fragment, calculate the distance to the light
  - If that distance is greater than the value in the light view depth map for that fragment's position, the light saw an occluder instead of the object, so the point is in shadow
  - If they are the same, no occlusion

Some More Details

- How do we find correspondence between light view depth map and fragment positions from the camera's point of view?
  - Multiply camera space coordinates by a matrix T such that $T = P_L M_L M_C^{-1}$
    - P_L is the light's projection matrix
    - M_L is the light's modelview matrix
    - M_C is the camera's modelview matrix
  - This translates a point in camera space into the light's clip space
  - But we have access to the world space coordinates so we can just use $P_L M_L$ to change from world space to light's clip space
  - Then just map clip space [-1, 1] to texture coordinates [0, 1] and you can index into the depth map

Some Tips on Multiple Render Targets

- For multiple render targets, set the nColorAttachments field of the framebuffer parameters to however many color attachments that you want
- When you want to use a framebuffer as a texture, use bindsurface(n) where n is the index of the color attachment you want to use

Multiple Render Targets

- You need to call glDrawBuffers() to set the color attachments to the proper render targets

```
GLenum buffers[n] = {GL_COLOR_ATTACHMENT0, ... ,
                     GL_COLOR_ATTACHMENTn };
glDrawBuffers(n, buffers);
```

- Call glBindFragDataLocation to tell the shader what render target to draw to
  - By default, it assumes that the first output is the first render target, second is the second render target, etc.

Multitexturing

- If you want to use multiple textures in a shader...

  ```
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(...);
  glActiveTexture(GL_TEXTURE1);
  glBindTexture(...);
  shader.bind(...);
  shader.setUniformValue("sampler0", 0);
  shader.setUniformValue("sampler1", 1);
  ```

- To clear textures you need to again call glActiveTexture before calling glBindTexture(GL_TEXTURE_2D, 0);