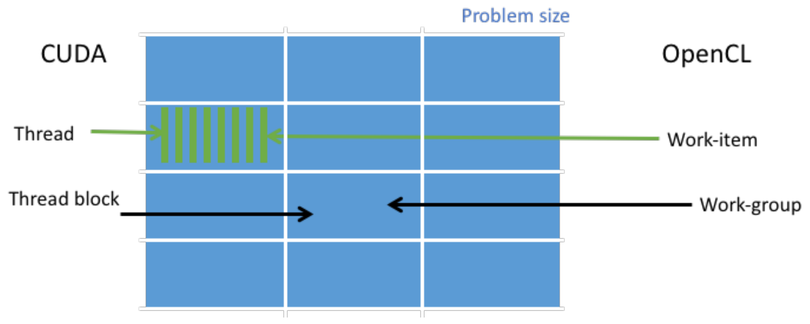
Lecture 12: CUDA and OpenCL by Comparison

- **Executional Model**



- **Kernels**

CUDA	OpenCL
Denote by __global__	Denote by __kernel
A function in the host code	Either a string (const char*), or read from a file
Compile with compilation of host code	Compile at runtime

- **Kernel Indexing**

CUDA	OpenCL
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

- **Enqueue a Kernel**

CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,

threads_per_block>>>();
```

Specify the number of thread blocks and threads per block

OpenCL C

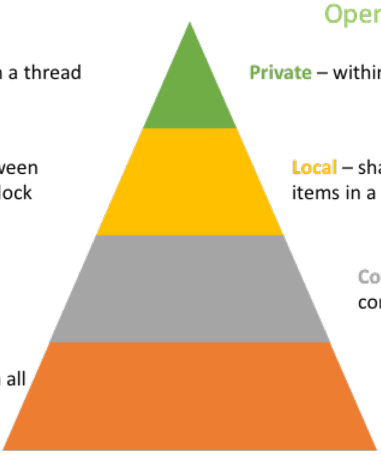
```
const size_t global[2] =
                 {300, 200};

const size_t local[2] =
                 {30, 20};

clEnqueueNDRangeKernel(
        queue, &kernel,
        2, 0, &global, &local,
        0, NULL, NULL);
```

Specify the problem size and (optionally) number of work-items per work-group

- **Kernel Synchronization**

CUDA	OpenCL
__syncthreads()	barrier()
__threadfenceblock()	mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
No equivalent	read_mem_fence()
No equivalent	write_mem_fence()
__threadfence()	Finish one kernel and start another

- **Memory Model**



CUDA	OpenCL
Local – within a thread	Private – within a work-item
Shared – shared between threads in a thread block	Local – shared between work-items in a work-group
Constant – a cache for constant memory	Constant – a cache for constant memory
Global – shared between all thread blocks	Global – shared between all work-groups

- **Memory Allocation and Copy**

	CUDA C	OpenCL C

Allocate	`float* d_x;`
	`cudaMalloc(&d_x,`
	`sizeof(float)*size);`
		`cl_mem d_x =`
		`  clCreateBuffer(context,`
		`    CL_MEM_READ_WRITE,`
		`    sizeof(float)*size,`
		`    NULL, NULL);`

Host to Device	`cudaMemcpy(d_x, h_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyHostToDevice);`
		`clEnqueueWriteBuffer(queue, d_x,`
		`     CL_TRUE, 0,`
		`     sizeof(float)*size,`
		`     h_x, 0, NULL, NULL);`

Device to Host	`cudaMemcpy(h_x, d_x,`
	`    sizeof(float)*size,`
	`    cudaMemcpyDeviceToHost);`
		`clEnqueueReadBuffer(queue, d_x,`
		`     CL_TRUE, 0,`
		`     sizeof(float)*size,`
		`     h_x, 0, NULL, NULL);`

Lecture 13: Introduction to OpenCL

- **What is OpenCL? Why do we use OpenCL?**
    - The Open Computing Language (OpenCL) is an open and royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors.
    - There are two benefits of using OpenCL
        - It can substantially accelerate parallel processing
        - It is portable in a cross-vendor fashion

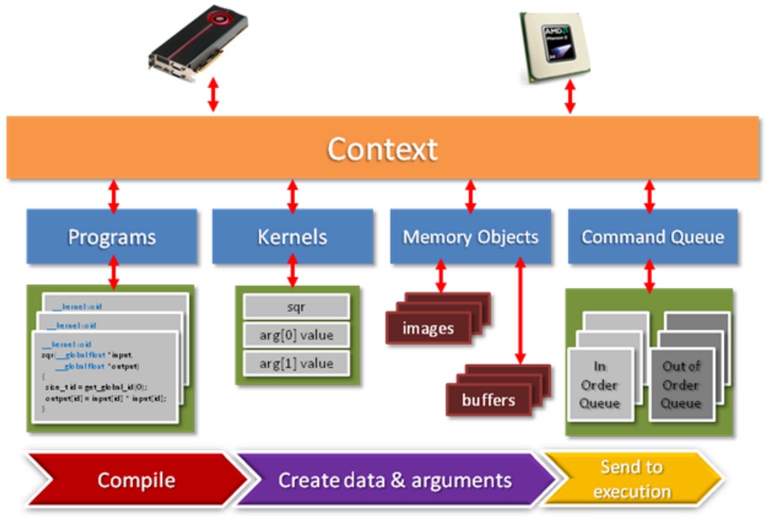
- **What is the Anatomy of OpenCL?**
    - Three main parts:
        - The **language specification** describes the syntax and programming interface for writing kernel programs that run on the supported accelerator.
        - The **platform API** gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices.
        - The **runtime API** uses contexts for managing objects such as command queues, memory objects, and kernel objects, as well as for executing kernels on one or more devices specified in the context.

- **What does OpenCL architecture look like?**
    - The OpenCL architecture is defined in four models:
        - **Platform model**: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices). (It is defined as a host connected to one or more OpenCL devices.)
        - **Execution model**: Defines how the OpenCL environment is configured on the host and how kernels are executed on the device. It comprises two components: kernels and host programs.
            - Kernels are the basic unit of executable code that runs on one or more OpenCL devices.
            - The host program is responsible for setting up and managing the execution of kernels on the OpenCL device through the use of context (The environment within which kernels execute and in which synchronization and memory management is defined). The host program executes on the host system, defines devices context, and queues kernel execution instances using command queues. Command queues accept three types of commands:
                - Kernel execution commands run the kernel command on the OpenCL devices.
                - Memory commands transfer memory objects between the memory space of the host and the memory space of the OpenCL devices.
                - Synchronization commands define the order in which commands are executed.
            - Kernels are queued in-order, but can be executed in-order or out-of-order. In in-order mode, the commands are executed serially as they are placed onto the queue. In out-of-order mode, the order the commands execute is based on the synchronization constraints placed on the command.

        - Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit.
    - **Memory model**: defines four regions of memory accessible to work-items when executing a kernel — global memory, constant memory, local memory, or private memory.
        - Global memory is a memory region in which all work-items and work-groups have read and write access on both the compute device and the host. It can be allocated only by the host during runtime.
        - Constant memory is a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.
        - Local memory is a region of memory used for data-sharing by work-items in a work-group. All work-items in the same work-group have both read and write access.
        - Private memory is a region that is accessible to only one work-item.
    - **Programming model**: Defines how the concurrency model is mapped to physical hardware.

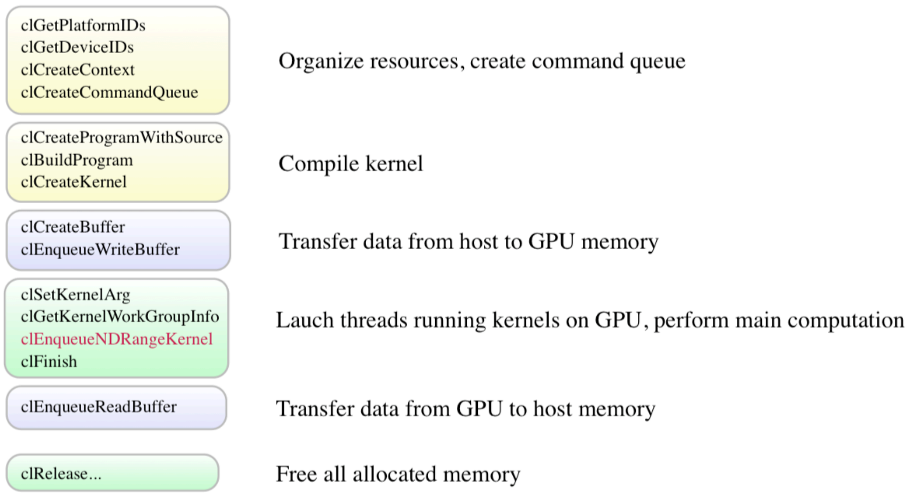
Lecture 14: (OpenCL) Programming Details

- **What is the structure of an OpenCL program?**
    - The OpenCL framework is divided into a platform layer API and runtime API. The **platform API** allows applications to query for OpenCL devices and manage them through a context. The **runtime API** makes use of the context to manage the execution of kernels on OpenCL devices. The figure below shows the basic structure of an OpenCL program.



```
__kernel void
sqr(__global float *input,
    __global float *output)
{
 size_t id = get_global_id(0);
 output[id] = input[id] * input[id];
}
```

- **What does the OpenCL program flow look like?**
  - Query the host system for OpenCL devices.
  - Create a context to associate the OpenCL devices.
  - Create programs that will run on one or more associated devices.
  - From the programs, select kernels to execute.
  - Create memory objects on the host or on the device.
  - Copy memory data to the device as needed.
  - Provide arguments for the kernels.
  - Submit the kernels to the command queue for execution.
  - Copy the results from the device to the host.
  - Release allocated objects.



Lecture 15: (OpenCL) Kernels, Memories, Synchronization and Events

- **OpenCL Kernels**
  - **__kernel qualifier** declares a function as a kernel, makes it visible to host code so it can be enqueued; Kernels can call other kernel-side functions.
  - A **work-item** is an instance of the kernel.
  - A **work-group** is a collection of work-item.
  - **Built-in work-item functions** can be used to query the information relating to the data that the kernel is asked to process. These functions allow querying of dimension size of the data, the global size for each dimension, the local work size, number of work-groups, as well the unique global and local work-item ID of the kernel that is being executed.
  - You specify total number of work-items (global dimensions), and **optionally** number of work-items per work-group (i.e., local dimensions)
  - **The local-size must divide evenly the global size in each dimension!**
  - A set of **vector data types** and **vector operations** were also added for easier manipulation of data set such as matrices. The vector data is defined with the type name follow by a value *n* that specifies the number of elements in the

vector. For example, a 4-component floating point vector would be float4. The supported components are: 2, 4, 8, and 16.

- There are several ways to access the components of a vector data type depending on how many components are in the vector. Vector components can also be accessed using numeric index to address the particular component. OpenCL also provides quick addressing to a grouping of components in a vector.
- Vector operations, such as +, -, *, /, &, |, etc., can be performed on each component in the vector independently.

- **OpenCL Memories**
  - **Address space**
    - **Global address space (__global)**
      - This address space refers to memory objects such as scalars, vectors, buffer objects, or image objects that are allocated in the global memory pool. For a GPU compute device, this is typically the frame buffer memory.
    - **Local address space (__local)**
      - This address space is typically used by local variables that are allocated in local memory. Memory in the local address space can be shared by all work-items of a work group. For example, local memory on a GPU compute device could be the local data store for one of the compute units (or core) on the GPU.
    - **Constant address space (__constant)**
      - This address space describes variables that allocated in global memory pool but can only be accessed as read-only variables. These variables are accessible by the entire global work-items. Variable that need to have a global scope must be declared in the constant address space.
    - **Private address space (__private)**
      - This address space describes variables that are passed into all functions (including __kernel functions) or variables declared without a qualifier. A private variable can only be accessed by the work-item in which it was declared.
    - The default address space for function arguments and local variables within a function or block is private, which means, all functions (including the __kernel function) and their arguments or local variables are in the __private address space. Arguments of a __kernel function declared as a pointer type can point to only one of the following memory spaces: __global, __local, or __constant.

  - **Memory objects**
    - **Buffer objects**
      - One-dimensional arrays in the traditional CPU sense and similar to memory allocated through malloc in a C program. Buffers can

contain any scalar data type, vector data type, or user-defined structure. The data stored in a buffer is sequential, such that the OpenCL kernel can access it using pointers in a random-access manner familiar to a C programmer.
        - Creating a buffer object requires a size, a context in which to create the buffer, and a set of creation flags. After creation, access to buffer objects is achieved through access functions, such as clEnqueueReadBuffer().
    - **Image objects**
        - Image objects exist in OpenCL to offer access to special function hardware on graphics processors that is designed to support highly efficient access to image data.
        - Images are multidimensional structures; and
        - opaque types that cannot be viewed directly through pointers in device code;
            - Image objects cannot be accessed through pointers on the device and cannot be both read and write within the same kernel.
            - It is accessed through specialized access functions in the kernel code, such as read_imagef(), read_imagei(), read_imageui(), write_imagef(), and so on.

- **Synchronization**
    - **Work-item synchronization**
        - The function **barrier(mem_fence_flag)** creates a barrier that blocks the current work-item until all other work-items in the same group has executed the barrier before allowing the work-item to proceed beyond the barrier.
        - The mem_fence_flag can be either CLK_LOCAL_MEM_FENCE, or CLK_GLOBAL_MEM_FENCE is use by the barrier function to either flush any variable in local or global memory or setup a memory fence to ensure that correct ordering of memory operations to either local or global memory.
        - It is important to note that when using barrier, all work-items in the work-group must execute the barrier function. If the barrier function is called within a conditional statement, it is important to ensure that all work-items in the work-group enter the conditional statement to execute the barrier.

    - **Kernel synchronization**
        - When a kernel is queued, it may not execute immediately. Execution can be forced by using a blocking call. The clEnqueueRead*() and clEnqueueWrite*() functions contain a blocking flag that, when set to CL_TRUE, forces the function to block until read/write commands have

completed. Using a blocking command forces the OpenCL runtime to flush all the commands in the queue by executing all kernels.

    - The recommended way to track the execution status of kernels in the command queue is to use events. Events allow the host application to work without blocking OpenCL calls. The host application can send tasks to the command queue and return later to check if the execution is done by querying the event status. This means that the command will not begin executing until all of the input events have completed.
    - All clEnqueue* functions take three arguments: The number of events to wait on, a list of events to wait on, and an event that the command creates that can be used by other commands to perform synchronization: clEnqueue*(...,num_events, events_wait_list, event_return);

- **Events**
  - In addition to using events in the clEnqueue functions to manage how kernels are executed, host application events can also be managed, allowing the application to better manage resources for maximum optimization.
    - clWaitForEvents(num_events, *event_list)
      - Blocks until events are complete
    - clEnqueueWaitForEvents(queue, num_events, *event_list)
      - Inserts a "WaitForEvents" into the queue
    - clEnqueueMarker(queue, *event)
      - Returns an event for a marker that moves through the queue
    - clGetEventInfo()
      - Command type and status: CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE, or error code
    - clSetEventCallback()
      - Called when command identified by event has completed
  - Profiling with events
    - clGetEventProfilingInfo()
    - Profiling time

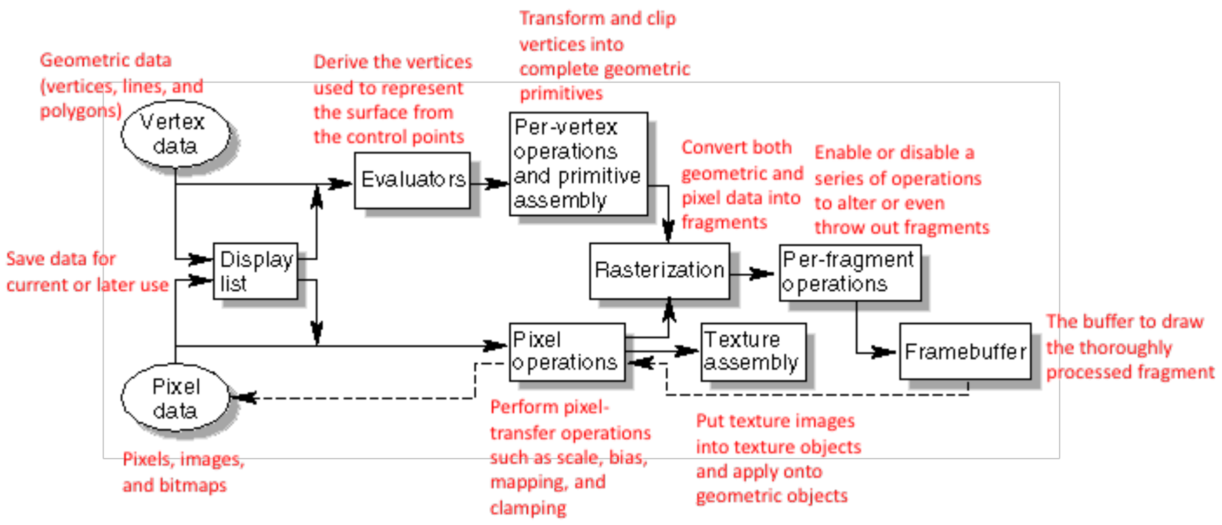
Lecture 16: Introduction to OpenGL

- **What is OpenGL?**
  - A cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.

- **What are the features of OpenGL?**
  - Contains more than 200 functions for building application programs.
  - The **primitive functions** define the elements that can produce images on the screen. These functions are of two types: geometric primitives, such as polygons, and discrete entities, such as bitmaps.
  - **Attribute functions** control the appearance of primitives. They define colors, line types, material properties, light sources, and textures.

- **Viewing functions** determine the properties of the camera.
- **Input functions** allows us to control the windows on the screen and to use the mouse and the keyboard.
- **Control functions** allow us to start and to terminate OpenGL programs and to turn on various OpenGL features.
- Programs written using OpenGL are portable to any computer that supports the interface. Use a simple toolkit, the OpenGL Utility Toolkit (GLUT) to interact with an operating system and the local windowing system.
- OpenGL is a state machine. State variables include current color of the objects, current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characteristics of lights, and material properties of the objects being drawn. The various states (or modes) that you put into then remain in effect until you change them. Many state variables refer to modes that are enabled or disabled with the command glEnable() or glDisable(). Each state variable or mode has a default value, and at any point you can query the system for each variable's current value.

- **What are the components of the OpenGL rendering pipeline?**
  - Geometric data, pixel data, display list, evaluators, per-vertex operations, primitive assembly, pixel operations, texture assembly, rasterization, fragment operations.

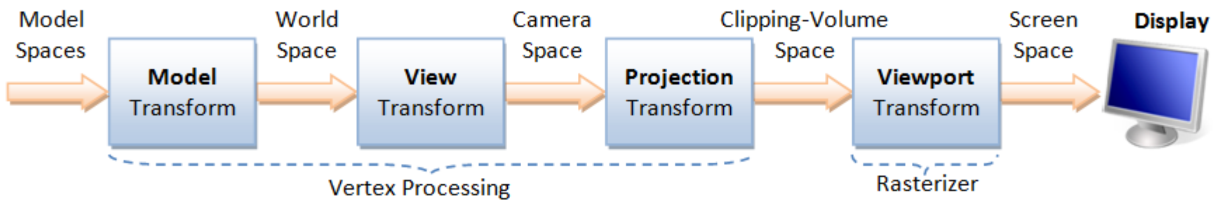

Geometric data (vertices, lines, and polygons)
Vertex data
Derive the vertices used to represent the surface from the control points
Evaluators
Transform and clip vertices into complete geometric primitives
Per-vertex operations and primitive assembly
Convert both geometric and pixel data into fragments
Enable or disable a series of operations to alter or even throw out fragments
Save data for current or later use
Display list
Rasterization
Per-fragment operations
Pixel operations
Texture assembly
Framebuffer
The buffer to draw the thoroughly processed fragment
Pixel data
Pixels, images, and bitmaps
Perform pixel-transfer operations such as scale, bias, mapping, and clamping
Put texture images into texture objects and apply onto geometric objects

- **OpenGL-Related Libraries**
  - **OpenGL core library (GL) #include<GL/gl.h>:** Contains all the required OpenGL function, e.g., glVertex3f()
  - **OpenGL Utility Library (GLU) (#include<GL/glu.h>:** Contains functions that are written using the functions in the core library but are helpful for users to have available, e.g., gluOrtho2D()
  - **OpenGL Utility Toolkit (GLUT) (#include<GL/glut.h>**: a library of functions that are common to virtually all modern windowing systems to simply operations

- Window management routines for windowing control
- Event loops and callback functions for interaction and animation

Lecture 17: (OpenGL) Transformations and Projections

In computer graphics, transform is carried by multiplying the vector with a transformation matrix. In the OpenGL rendering pipeline, there are four types of transforms:



- **Modeling and viewing transformations**: orient the model and the camera relative to each other to obtain the desired final image
  - **Model transform** consists of scaling, rotation, and translation. OpenGL functions: glScale, glRotate, and glTranslate. Successive transforms can be combined before applying to the vector.
  - **View transform** positions the camera onto the world space by specifying three view parameters: EYE (location of the camera), AT (direction the camera is aiming at), and UP (the upward orientation of the camera). OpenGL function: gluLookAt().
- **Projection transformations**: specify the shape and orientation of the viewing volume, which determines how a scene is projected onto the screen (with a perspective or orthographic projection) and which objects or parts of objects are clipped out of the scene.
  - **The perspective projection** sets the viewing volume as a frustum of a pyramid. OpenGL has a GLU function gluPerspective to choose the perspective projection, and a GL function glFrustrum to set clipping volume.
  - **The orthographic projection** creates an orthographic parallel viewing volume. OpenGL functions: glOrtho(), gluOrtho2D().
- **Viewport transformation**: control the conversion of three-dimensional model coordinates to screen coordinates
  - OpenGL provides glViewPort() to define a pixel rectangle in the window into which the final image is mapped, and glDepthRange to set the z-range of viewport.

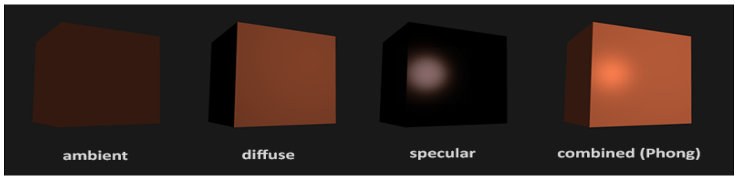
Manipulating the matrix stacks: save and restore certain transformations with glPushMatrix() and glPopMatrix().

Lecture 18: (OpenGL) Lights and Materials

- **Lights**
  - In OpenGL, light models are developed to capture the core effect that light has when it falls on objects and makes them visible.
  - The OpenGL lighting model are based on the physics of light. One of those models is called the **Phong lighting model**. Phong lighting model is a local

illumination model, which is compute inexpensive and extensively used especially in the earlier days. It considers four types of lightings: diffuse, specular, ambient and emissive.

- **Ambient light** is a constant amount of light applied to every point of the scene.
- **Diffuse light** models distant directional light source. It makes the parts of objects that face the light brighter than the parts that are opposite from it.
- **Specular lighting** adds the position of the viewer. It is more a property of the object, rather than the light itself. It makes parts of things shine when light hits them at a very specific angle and the viewer is positioned at a specific point.



- **OpenGL Lighting**
    - OpenGL provides point sources (omni-directional), spotlights (directional with cone-shaped), and ambient light (a constant factor). Light source may be located at a fixed position or infinitely far away. Each source has separate ambient, diffuse, and specular components. Each source has RGB components. The lighting calculation is performed on each of the components independently. Manipulated by **glLight**().

- **Materials**
    - In the real world, each object reacts differently to light. If we want to simulate several types of objects in OpenGL, we have to define material properties specific to each object.
    - Material Colors
        - Like lights, materials have different ambient, diffuse, and specular colors, which determine the ambient, diffuse, and specular reflectance of the material.
        - In addition to ambient, diffuse, and specular colors, materials have an **emissive color**, which simulates light originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intensity to the object, but is unaffected by any light sources.
    - Materials can be modeled using **glMaterial**() function to specify these attributes for the front (GL_FRONT), back (GL_BACK), or both (GL_FRONT_AND_BACK) surfaces. The front face is determined by the surface normal (implicitly defined by the vertices with right-hand rule, or glNormal() function).
    - glColorMaterial():
        - Another technique for minimizing performance costs associated with changing material properties is to use glColorMaterial(). It causes the material property (or properties) specified by *mode* of the specified material face (or faces) specified by *face* to track the value of the current

color at all times. A change to the current color (using glColor*()) immediately updates the specified material properties.

Lecture 19: (OpenGL) Bitmaps, Images, and Pixels

- **What is Bitmap?**
    - A bitmap is a rectangular array of 0s and 1s that serves as a drawing mask for a corresponding rectangular portion of the window.

- **How to draw character in Bitmap?**
    - Set the Current Raster Position by calling glRasterPos*()
    - Call glBitmap() command to draw a single bitmap on the screen.
    - Choose a Color for the Bitmap

- **What is image?**
    - An image is similar to a bitmap, but instead of containing only a single bit for each pixel in a rectangular region of the screen, an image can contain much more information.

- **What OpenGL commands are available to manipulate image data?**
    - Three basic commands that manipulate image data:
        - **glReadPixels**() - Reads a rectangular array of pixels from the framebuffer and stores the data in processor memory.
        - **glDrawPixels**() - Writes a rectangular array of pixels from data kept in processor memory into the framebuffer at the current raster position specified by glRasterPos*().
        - **glCopyPixels**() - Copies a rectangular array of pixels from one part of the framebuffer to another. This command behaves similarly to a call to glReadPixels() followed by a call to glDrawPixels(), but the data is never written into processor memory.

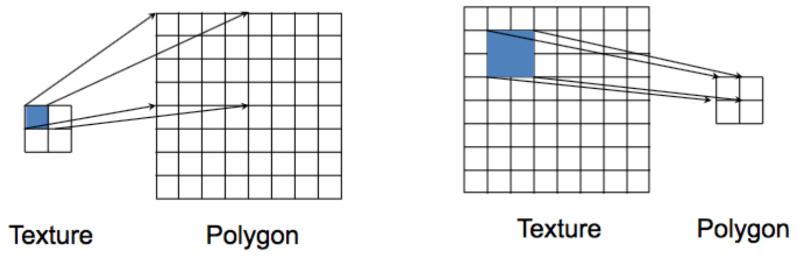
Lecture 20: (OpenGL) Textures

- **What is texture mapping, why do we need it?**
    - Texture mapping combines pixels with geometric objects to provide images of seemingly great complexity but without the overhead of building large geometric models

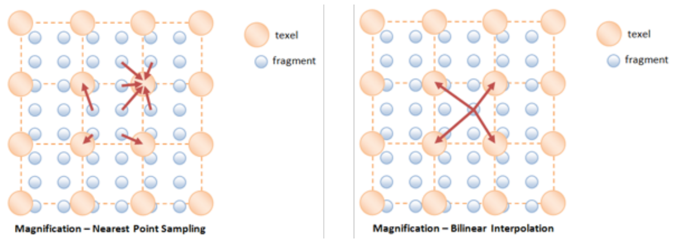
- **What is the process of texturing mapping?**
    - A small area of the texture pattern maps to the area of the geometric surface, corresponding to a pixel in the final image. There are three basic steps:
        - Specify a texture image and place it in memory using glTexImage2D()
        - Assign texture coordinates to vertices by glTexCoord*()
        - Specify texture parameters such as wrapping and filtering by using glTexParameteri()

- **What is mipmap?**
  - A series of prefiltered texture maps of decreasing resolutions, called mipmaps. When using mipmapping, OpenGL automatically determines which texture map to use based on the size (in pixels) of the object being mapped. With this approach, the level of detail in the texture map is appropriate for the image that's drawn on the screen - as the image of the object gets smaller, the size of the texture map decreases. Mipmapping requires some extra computation and texture storage area; however, when it's not used, textures that are mapped onto smaller objects might shimmer and flash as the objects move.

- **Magnification and Minification**
  - Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel (magnification) to a large collection of texels (minification)



  - Two commonly used filtering methods for magnification and minification are:
    - **Nearest point filtering**: the texture color-value of the fragment is taken from the nearest texel
    - **Bilinear interpolation**: the texture color-vlaue of the fragment is formed via bilinear interpolation of the four nearest texels


texel
fragment
Magnification – Nearest Point Sampling
texel
fragment
Magnification – Bilinear Interpolation

    - We can use **glTexParameteri**() to specifiy the magnification and minification filtering methods.

==
(The following content is NOT going to be concluded in your final exam, but I'm giving it to you anyway for your reference.)

Lecture 21: (CUDA) Graphics Interoperability

- **What is Graphics Interoperability?**
  - Graphics interoperability is the functionality to interact between general-purpose computation and rendering modes, makes it possible to compute and visualize the same data

- **How does CUDA-OpenGL interoperation work?**
    - CUDA used for data generation, calculation, image manipulation
    - OpenGL used to draw pixels of vertices on the screen
    - They share data through common memory in the framebuffer

- **CUDA-OpenGL Interoperation steps?**
    - Step 1: decide what data CUDA will process
    - Step 2: allocate with OpenGL by glGenBuffers(), glBindBuffer(), and glBufferData()
    - Step 3: register with CUDA by cudaGraphicsGLRegisterBuffer()
    - Step 4: map buffer to get CUDA pointer by cudaGraphicsMapResources()
    - Step 5: pass pointer to CUDA kernel by cudaGraphicsResourdeGetMappedPointer()
    - Step 6: release pointer to invoke kernel function and unmap resources by cudaGraphicsUnmapResources()
    - Step 7: use result in OpenGL graphics

Lecture 22: (OpenCL) Graphics Interoperation

- **OpenCL/OpenGL Interop**
    - OpenCL is specifically crafted to increase computing efficiency across platforms, and OpenGL is a popular graphics API. Using OpenCL to manipulate OpenGL objects has important advantages: the GPU is usually faster and data transfer from Host memory to Device memory is kept to a minimum.
    - **True (zero-copy) interoperability** is about passing ownership between the APIs and not the actual data of a resource. It is important to remember that it is an OpenCL memory object created from an OpenGL object and not vice versa:
        - OpenGL texture (or render-buffer) becomes an OpenCL image (via clCreateFromGLTexture).
        - OpenGL (vertex) buffers are transformed to OpenCL buffers in a similar way (clCreateFromGLBuffer).

- **Interop Modes**
    - Direct OpenGL texture sharing via **clCreateFromGLTexture**.
        - This is the most efficient mode of performance for interoperability with an OpenCL device. It also allows the modification of textures “in-place.” The number of OpenGL texture formats and targets that are possible to share via OpenCL images is limited.
    - Creating an intermediate (staging) Pixel-Buffer-Object for the OpenGL texture via **clCreateFromGLBuffer**, updating the buffer with OpenCL, and copying the results back to the texture.
        - The downside of this approach is that even though the PBO itself does support zero-copy with OpenCL, the final PBO-to-texture transfer still

        relies on copying, so this method is slower than the direct sharing method introduced above.
        - The upside is that there are fewer restrictions on the texture formats and targets beyond those imposed by the glTexSubImage2D, and allows for the potential of sharing textures of more formats.
    - Mapping the GL texture with **glMapBuffer**, wrapping the resulting host pointer as an OpenCL buffer for processing, and copying the results back on **glUnmapBuffer**.
        - Slow (the OpenCL buffer is created/released in every frame)

- **Steps for Interop Programming**
    - 1. Verify GL sharing is supported
        - Get extension string for CL_DEVICE_EXTENSIONS by calling clGetDeviceInfo()
        - Check for the appropriate platform extension string: cl_khr_gl_sharing
    - 2. Set up a shared context (to use GL devices in OpenCL context)
        - Use platform context properties to indicate GL interop
        - Create an OpenCL context using active GL devices
    - 3. Sharing data (using GL objects in OpenCL)
        - Create objects in GL like normal
        - Create reference object in OpenCL
            - clCreateFromGLBuffer
            - clCreateFromGLTexture2D
            - clCreateFromGLTexture3D
            - clCreateFromGLRenderbuffer
        - Switch ownership from GL to OpenCL to use
        - Release reference in OpenCL first then destroy in GL
    - 4. Execution
        - Synchronizing zccess to shared OpenCL/OpenGL objects
            - **glFinish**
            - **clEnqueueAcquireGLObjects**
            - **clEnqueueReleaseGLObjects**
            - **clFinish**