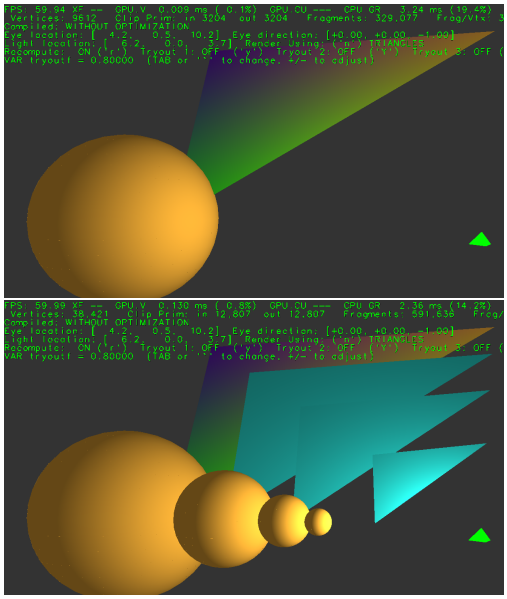

Problem 0: Follow the instructions on the `https://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show the sphere from the `demo-05-more-vulkan.cc` code, see the upper screenshot on the right.

User Interface

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene, but they can be set to move the sphere or the light. Press `l` to move the light around, `b` to move the sphere (ball), and `e` to move the eye (which is what the arrow keys do when the program starts). Pressing `Shift` and an arrow key will move by a larger distance (than if `Shift` were not pressed) and pressing `Ctrl` and an arrow key will move by a smaller distance. The eye can be aimed up and down by pressing `Home` and `End` and the eye can be rotated by pressing `Insert` and `Delete`.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size.

Display of Performance-Related Data

The top green text line shows performance in various ways. The number to the right of FPS shows the frame rate, the number of frames completed per second. On some displays 60 is the target frame rate and anything significantly lower than that indicates mediocre performance.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows how long the computational accelerator takes per frame. The computational accelerator computes physics in some assignments, but not this one and so the time should be shown as `---`. On some of the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends recording graphics commands (or whatever it does in the callback installed by `vh.cbs_cmd_record.push_back`. CPU PH is the amount of time that the CPU spends on physics or whatever it does in the callback installed by the call to `vh.display_cb_set`.

The second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out). The next line indicates whether the code was compiled with optimization. Use the version without optimization for debugging and the version with optimization for performance measurements.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, and variables that will be needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` cycles through the different variables. To locate variables which can be set this way, and to see how they were set, search for `variable_control.insert` in the assignment file.

Code Generation and Debug Support

The compiler generates two versions of the code, `hw01` and `hw01-debug`. Use `hw01` to measure performance, but use `hw01-debug` for debugging. The `hw01-debug` version is compiled with optimization turned off and with Vulkan validation turned on. You are strongly encouraged to run `hw01-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When Vulkan validation is on (which is currently in both debug and normal versions) helpful error and warning messages will be printed about misuse or abuse of the Vulkan API. These will appear on the terminal window (which might be a gdb session) from which `hw01-debug` was started. Currently, the code is set so that all warnings are fatal (except for a select few, which won't generate messages).

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In most assignments these include the variables `opt_tryout1`, `opt_tryout2`, `opt_tryout3`, and `opt_tryoutf`. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Overview of the Vulkan Helper Classes

The Vulkan Helper Classes are a set of classes and other C++ code intended to make Vulkan easier to learn and use for students in LSU EE 4702 (GPU Programming). Its development started in the summer of 2021 and it continues to evolve rapidly, so this description may well be obsolete after the 2021 Homework 1 assignment deadline.

Using the Vulkan Helper Code in this Assignment

For this assignment it is important to understand how to use the `VVertex_Buffer_Set` objects to specify vertices for a draw, and how to specify transformation matrices using `VTransform`. These classes are described below with more than enough detail to solve this assignment. In addition to the descriptions in this handout, look for the comments in the `render` routine.

For this discussion one needs some understanding of what a rendering pipeline is and of transformations and coordinate spaces used in graphics.

The code in `hw01.cc` has a routine named `render` in which most of the solution is to be placed. The `render` routine is called each time the window needs to be updated, which on many computers will be 60 times a second.

The render routine sets up two rendering pipelines, one for the triangle, `pipe_lonely`, and one for the sphere, `pipe_sphere`, and prepares a set of vertices for each, `bset_lonely` and `bset_sphere`. It also prepares transformation matrices so that the triangles formed by the vertices in the two vertex sets are placed in the desired place in the window. Most importantly it records draws using each pipeline with its vertex set. See the following sections to understand how that is done.

In this assignment it will be necessary to add new vertices forming new triangles (Problem 1) and to use transformation matrices so that additional spheres can be rendered by re-using the existing sphere coordinates.

Using Vertex Buffer Sets

The vertex buffer set class, `VVertex_Buffer_Set`, manages the vertices that will be streamed into a rendering pipeline. Objects of this class by convention have names that start `bset`. In the Homework 1 assignment there are two such objects `bset_lonely` and `bset_sphere`. Here, *vertex* refers to a collection of attributes associated with a part of a geometric object. Typical attributes are coordinate, normal, and color.

The first step in using a vertex buffer set object is to reset it. A vertex buffer set object is reset by calling the `reset` member function with a pipeline as an argument, for example `bset_sphere.reset(pipe_sphere);`. This will clear any vertex attributes that the object is carrying and will also set the object to expect only those attributes needed by the pipeline given in the argument. For example, if `pipe_sphere` in the example above does not use texture coordinates (and it doesn't in Homework 1) then `bset_sphere` will not expect texture coordinates and will exit with an error if an attempt is made to insert texture coordinates.

After being reset a vertex buffer set is ready for new vertex attributes to be inserted. This is done using the insertion operator, `<<`. Each attribute has a different type and so there is no need to specify which attribute is being inserted. For example, in the code below ...

```
bset_lonely.reset(pipe_lonely);
pCoor p0 = { 0, 0, 0 };
pNorm tri_norm = cross( p0, p1, p2 );
pColor color_tri( .470, .553, .965 ); // Red, Green, Blue
bset_lonely << color_tri << tri_norm << p0;
```

... three different attributes are being appended to `bset_lonely`, a vertex coordinate (`p0`), a vertex normal (`tri_norm`), and a color (`color_tri`). After execution of the code above `bset_lonely` holds one vertex and the vertex has three attributes. The three attributes could have been inserted in any order (relative to each other) and using one to three statements. For example, the following are valid ways of inserting two more vertices:

```
pCoor p1 = { 9, 6, -4 };
pCoor p2 = { 0, 5, -3 };
bset_lonely << p1 << color_lsu_spirit_gold << tri_norm;
bset_lonely << color_lsu_spirit_purple;
bset_lonely << tri_norm;
bset_lonely << p2;
```

After the execution of the second code fragment above `bset_lonely` holds three vertices, each with three attributes.

Currently a vertex buffer set object can manage six attributes: vertex coordinates (type `pCoor` in member `pos`), colors (type `pColor` in member `color`), normals (type `pVect4` in member `normal`), texture coordinates (type `pTCoor` in member `tcoord`), scalar integers (type `int` in member `int1`), and 2-element integer vectors (type `ivec2` in member `int2`). The first four are used by the default Vulkan Helper shaders and correspond to the vertex attributes defined by OpenGL. The `int1` and `int2` attributes can only be used with custom shaders, and those shaders will know how to interpret those integers. (The member names above may change by Homework 2, for example, `pos` may be changed to `coor` and `tcoord` may change to `tcoor`.)

The insertion operation appends the attribute to an array on the host (a C++ std vector). To copy the attributes to the device (GPU) use the member function `to_dev()`, for example, `bset_lonely.to_dev();`. The vertex buffer set object does not keep track of changes to the attribute arrays, and will copy the arrays to the device even if no changes have been made since the last copy. That is, `bset_lonely.to_dev(); bset_lonely.to_dev();` will copy the attributes twice. This behavior may change by Homework 2.

The `to_dev()` member function copies data immediately. That is in contrast to other code in the `render` routine that records commands to a command buffer for later execution.

The purpose of a vertex buffer object is to provide inputs to a pipeline. A `VPipeline` object's `record_draw` member function takes a vertex buffer set object as a parameter and records a draw command in which the buffer set is streamed into the pipeline. For example, `pipe_lonely.record_draw(cb, bset_lonely);` records a draw command (actually several setup commands followed by a draw command) in which pipeline `pipe_lonely` is sent vertex attributes from `bset_lonely`. The commands

recorded into `cb` will be executed after `render` returns, so `bset_lonely` should not be changed until the next call to `render`.

The sequence of commands `bset_lonely.reset(pipe_lonely);`, `bset_lonely << color_green << tri_norm << p0;`, `bset_lonely.to_dev();` only need to be executed when the vertices change. There is no need to re-execute them on later calls to `render` if the exact same vertex attributes would be inserted. In the homework assignment package they *are* re-executed every time `render` is called, which is wasteful, but something that is to be fixed (for the sphere) in the assignment.

The assignment code also calls member function `destroy()`. This frees resources consumed by the vertex buffer set object. This should be called when the object is no longer needed, especially in long-running programs in which the GPU can become stuffed with no-longer-needed resources.

Using Transformation Matrices

Transformation matrices are used by the rendering pipeline's shaders to transform vertex coordinates from one space to another. The shaders in this assignment are provided as part of the Vulkan Helper code, and so the transformation matrices must be in the form that they expect. The `VTransform` class holds a set of transformation matrices and allows them to be conveniently changed.

In a correct solution to Problem 2 the transformation matrix used in draws of the sphere will have to be changed. That is done using the `VTransform::global_from_local_set_for` member function, so keep reading.

The `VTransform` class recognizes four coordinate spaces, *local*, *global*, *eye*, and *clip*. The *Local* space coordinate space is one that is convenient for a particular object. For example, the sphere is constructed in a local coordinate space in which the center is at the origin and the radius is one. The *Global* space is one that is shared by all parts of a scene. In this assignment the triangle coordinates and eye coordinates are specified in global space. In the *eye* coordinate space the eye is at the origin and is looking in the $-z$ direction. Eye space historically was used for lighting calculations, and is the space in which perspective projection transforms are defined. In *clip* space the coordinates of all visible items are in a cube with vertices at ± 1.

Object space is another term used when describing rendering pipelines. Object space is defined as the coordinate space of the vertices used as input to a rendering pipeline. That can be either local or global space using the definition above.

The `VTransform` keeps track of three transformations, *eye from global*, *clip from eye*, and *global from local*. The eye from global transform is chosen based on the location of the eye in global space and the direction it is looking. It is set using the `eye_from_global_set` member function. The clip from eye transform is chosen based on the position and size of the user's monitor in eye space (to achieve a perspective projection), it is set using the `clip_from_eye_set` member function. These should be set once during a render pass. The code below sets these two transformations using the UI-modifiable `eye_location` and `eye_direction` variables and the aspect ratio of the window Vulkan is rendering into (obtained ultimately from the window manager):

```
transform.eye_from_global_set
  ( pMatrix_Rotation(eye_direction,pVect(0,0,-1))
    * pMatrix_Translate(-eye_location) );

const int win_width = vh.get_width(), win_height = vh.get_height();
const float aspect = float(win_width) / win_height;

transform.clip_from_eye_set
  ( pMatrix_Frustum(-.8,  .8,    -.8/aspect,  .8/aspect, 1,    5000) );
       // Frustum: left, right, bottom,      top,       near, far
```

(In the actual assignment code the clip from eye transform also flips the y axis direction. In later assignments the y axis will point up without such a trick.)

Before recording a draw command, a pipeline needs to be given a transform. (That means, it needs to be given a `vk::Buffer` handle of a uniform object that will contain the transform matrices.) For pipelines using a global transform that is done using the `VTransform::use_for` member function called with a pipeline as an argument, for example, `transform.use_for(pipe_lonely);`. This only needs to be done once per rendering pass, since the global transform does not change. (It could even be done once period, if `VTransform` preserves the buffer, which it might one day do.)

The local space can be changed multiple times in a rendering pass. In the unmodified assignment it is changed once, for the sphere. The solution to problem 2 would have the local space modified multiple times.

To specify that a pipeline is to use a local transformation matrix call the `VTransform::global_from_local_set_for(XFORM,PIPE)` member function. The first argument is the transformation matrix, the second is a `VPipeline` object. For example,

```
pMatrix glo_from_loc =
  pMatrix_Translate( sphere_location ) * pMatrix_Scale(sphere_radius);
transform.global_from_local_set_for( glo_from_loc, pipe_sphere );
```

scales the sphere so that it is radius `sphere_radius` and moves it to location `sphere_location`. Each time `global_from_local_set_for` is called (in the first render) it will create a new Vulkan `vk::Buffer` to hold the transformation. So if it is called 10 times in a render pass it will create 10 buffers, but those buffers will be re-used in the next pass. Calling it thousands of times will likely exhaust storage, meaning that other methods of managing local space are needed.

Using Pipelines

The pipeline class, `VPipeline`, manages Vulkan rendering pipeline objects and information needed to use them to draw. In this assignment there will be no need to modify the pipelines themselves, but that will change in future assignments. This brief description is provided for background and to provide a better overall understanding of what the code is doing.

A Vulkan pipeline object consists of a sequence of fixed-function and programmable stages along with information on the resources to be accessed including vertex shader input types, uniform types, and the type of frame buffer it expects to access. To use a pipeline object the pipeline and the data it access must all be *bound* (identified, sort of like specifying call arguments in advance) and then a Vulkan draw command is recorded. The argument to the draw itself is little more than the number of vertices. The vertices come from the bound vertex buffers. A `VPipeline` object records a draw using the `record_draw` member function. This function automatically binds the vertex buffers, pipeline, and other resources. In this assignment see, for example, `pipe_sphere.record_draw(cb, bset_sphere);`.

A `VPipeline` object can be set to use user-provided shaders or it can use its own shaders, the latter of which is done by the Homework 1 code. A `VPipeline` object must be created before use. Pipeline creation starts with a call to the `init` member function, followed by calls to functions setting options, specifying shaders, specifying data to use, and so on, creation is completed with a call to the `create` member function. Most of these member functions return a reference to the object, so they can be conveniently stringed together, for example:

```
pipe_sphere
  .init( vh.qs )
  .use_uni_light( uni_light )
  .lighting_on()
  .topology_set( prim_want )
  .create( vh.qs.render_pass );
```

A pipeline object when cast to a Boolean (as in `if ( !pipe_sphere )`) will return false until it is created. Options such as the topology (how to group vertices into primitives) can only be set before the pipeline is created. The only way to change them is to call `destroy` and then re-create the pipeline. This is time consuming and so should not be done frequently.

The code for this assignment is based on the `demo-05-more-vulkan` code used in class which shows how to render a sphere and a triangle. Routine `render()` renders the sphere and triangle—the lonely triangle.

Problem 1: Modify the code in `render` so that the lonely triangle has `opt_n_objects` friends, where `opt_n_objects` is the name of a user-interface modifiable variable. Those new triangles should be placed between the lonely triangle and the light so that each vertex of a new triangle is on a line between an original vertex and the light. See the second screenshot on the first page of the assignment. The color of the new triangles should be different than the lonely triangle (or any of the lonely triangle's vertices).

The coordinates of the lonely triangle vertices are in objects `p0`, `p1`, and `p2` and for your convenience have been placed in vector `tri_vtx`. The coordinates of the light are in object `light_location`.

Use `bset_lonely` for the shader inputs; it should hold shader inputs for the lonely triangle (as it does before any changes are made) along with the shader inputs for the new triangles. For this problem there is no need to change `pipe_lonely` or how it is used.

- Compute the coordinates of the new triangles and insert them into `bset_lonely`.
- Compute the normal of the new triangle and insert that into `bset_lonely`.
- Also insert the new colors.
- The number of coordinates inserted into `bset_lonely` must match the number of colors and normals.

To help debug your code it might be helpful (or at least mildly interesting) to move the light (using the UI) and to change the value of `opt_n_objects`.

Problem 2: Modify the code so that `opt_n_objects` new spheres are rendered. The centers of the new spheres should be on a line between the center of the original sphere and the light. The size of the new spheres should be smaller than the original (though the exact size is not important) and the spheres should be positioned so that they exactly touch but do not intersect. See the second screenshot on the first page of this assignment.

Unlike the triangles, each new sphere should be rendered by reusing the shader inputs used to render the original, `bset_sphere`, and by reusing the rendering pipeline `pipe_sphere`. To place the a new sphere specify a new local-to-global transformation matrix and record a new pipeline draw. See the use of `transform.global_from_local`.

Problem 3: The code in `demo-05-more-vulkan` and this assignment is actually quite wasteful. In fact, more experienced programmers might cringe just looking at it. I am referring in particular to the code computing the sphere coordinates. The problem is that those coordinates are computed each frame even when the sphere does not change. (The sphere can change if the value of `slices` is changed using the user interface.)

(*a*) Modify the code in `render` and elsewhere so that when `opt_recompute` is false the sphere is only recomputed when necessary. Keep in mind that `bset_sphere` will hold its contents until reset, and the data sent to the device using `bset_sphere.to_dev()` will stay there until removed, so the data sent for rendering one frame can be used to render the next frame.

(*b*) Investigate the impact on performance by switching `opt_recompute` on and off and noting its effect on performance. Note that the time needed to execute `render` is shown to the right of `GPU GR` in the green text.

- Indicate that GPU being used. (That can be found in the program output to the terminal near the top. Look for a line starting `Props driver`.
- Does optimized code make a big difference?
- In optimized code, who many slices does it take to degrade frame rate with and without re-computation?