Graphic Programming

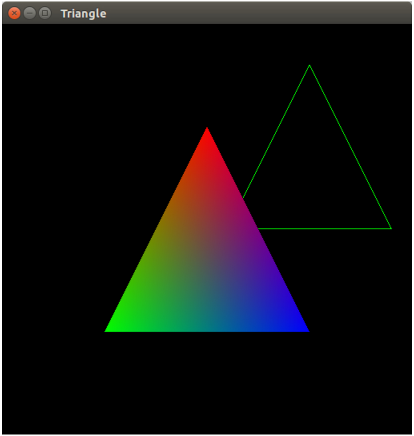
Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

OpenGL As A Tool

- We will use OpenGL as a tool to view results of transformation, viewing effect, lighting effect, etc
- In doing so, we can focus on more important aspect of computer graphics
  - No need to worry about hidden-surface removal
  - No need to worry about Windowing systems
- OpenGL is a C library
  - **NOTE** that libraries provided by the textbook were written in C++
  - The `initShader()` function written in C will be provided
  - You should be able to implement other functions in C by yourself
- So, we only need to know enough about OpenGL for our purpose

Simple OpenGL Example

- In this section, we are going to look at our first OpenGL program as shown below:



Programmable Graphic Pipeline

- In this course, we will use programmable pipeline instead of fixed-function pipeline
- We need two additional programs written in OpenGL Shading Language (GLSL)
- A GLSL program is a C-style program with additional syntax
- We will program two stages of the rendering pipeline
  - Vertex Shader
    - Process individual vertices
    - Transformation
    - Projection
    - Pre-vertex lighting
  - Fragment Shader
    - Process fragment generated by the rasterization
    - Colors of fragments
    - Depth values of fragments

Programmable Graphic Pipeline

- Vertex Shader (`vshader.glsl`)

```
#version 130

in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

- Fragment Shader (`fshader.glsl`)

```
#version 130

in vec4 color;
out vec4 fColor;

void main()
{
    fColor = color;
}
```

Programmable Graphic Pipeline (Older Version/OSX)

- Vertex Shader (`vshader.glsl`)

```
#version 120

attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

- Fragment Shader (`fshader.glsl`)

```
#version 120

varying vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Loading Shader Programs

- Shader programs (vertex and fragment) must be loaded into the rendering pipeline
- The `initShader()` function written in C is provided in `initShader.h` and `initShader.c`
- Simply include `initShader.h` and call the `initShader()` function

```
#include "initShader.h";
:
:
    GLuint program = initShader("vshader.glsl", "fshader.glsl");
    glUseProgram(program);
:
```

- Make sure to include `initShader.c` or `initShader.o` during compilation

Include Necessary Headers

- We need to include a number of header files as shown below:

```
#ifdef __APPLE__  // include Mac OS X verions of headers
#include <OpenGL/OpenGL.h>
#include <GLUT/glut.h>
#else             // non-Mac OS X operating systems
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <GL/freeglut_ext.h>
#endif

#include "initShader.h";
```

- `OpenGL` (Open Graphics Library) and `glew` (OpenGL Extension Wrangler Library) are cross-platform libraries for OpenGL
- `glut` (OpenGL Utility Toolkit) is a library of utilities for OpenGL which primarily focuses on window definition, window control, and monitoring of keyboard and mouse input

The `main()` Function

- The `main()` function of this program is as follows:

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(512, 512);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Triangle");
    glewInit();                         // OSX - Remove this line
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();

    return 0;
}
```

The main() Function

- `glutInit(&argc, argv);`

  Initialize the GLUT library where options can be passed via command line

- `glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);`

  Initialize the display mode

  - `GLUT_RGBA`: Red, Green, Blue, and Alpha (RGBA) color mode
  - `GLUT_DOUBLE`: Double buffer (one for displaying and another for rendering)
  - `GLUT_DEPTH`: Enable depth buffer (hidden surface removal)

- `glutInitWindowSize(512, 512);`

  Initialize window size to 512 pixels (width) by 512 pixels (height)

The `main()` Function

- ```
  glutInitWindowPosition(100, 100);
  ```
  Initialize the position of the top-left corner of the window
- ```
  glutCreateWindow("Triangle");
  ```
  Create the window with "Triangle" as its title
- ```
  glewInit();
  ```
  Initialize the OpenGL extension entry points. This function must be called before calling any core OpenGL functions.
  ```

The `main()` Function

- ```
  init();
  ```
  - A user-defined function
  - Define vertices of objects
  - Upload vertices and their attribute to GPU memory
  - Set uniform variables for passing data between CPU application and GPU application (shader programs)
- The `glutDisplayFunc()` Function:
  - Signature:
    ```
    void glutDisplayFunc(void (*func)(void));
    ```
  - Example:
    ```
    glutDisplayFunc(display);
    ```
  - Sets the display callback function.
  - The `display()` function will be called every time the current window needs update
  - The signature of the `display()` function must be
    ```
    void display(void);
    ```
    ```

The `main()` Function

- The `glutKeyboardFunc()` Function
  - Signature:

```
void glutKeyboardFunc(void (*func)(unsigned char key,
                                   int x, int y));
```

  - Example:

```
glutKeyboardFunc(keyboard);
```

  - Sets the keyboard events callback function.
  - The `keyboard()` function will be called when there is an event generated by the keyboard.
  - The signature of the `keyboard()` function must be

```
void keyboard(unsigned char key, int x, int y);
```

-
```
glutMainLoop();
```
  - Enters the GLUT event processing loop

The keyboard() Function

- We are going to start with a very simple keyboard() function:

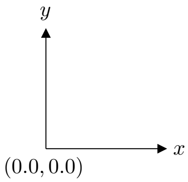
```
void keyboard(unsigned char key, int mousex, int mousey)
{
    if(key == 'q')
        exit(0);

    glutPostRedisplay();
}
```

- This function will be called every time a key is pressed
- Arguments are set by the system:
  - key: A character associated with the pressed-key
  - mousex: the x coordinate of the mouse pointer when the even is generated
  - mousey: the x coordinate of the mouse pointer when the even is generated

OpenGL Axes

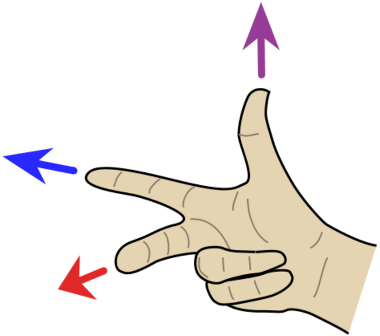
- In OpenGL,
  - the positive direction of the x-axis goes to the right, and
  - the positive direction of the y-axis goes up:



- Unit type is a floating-point number

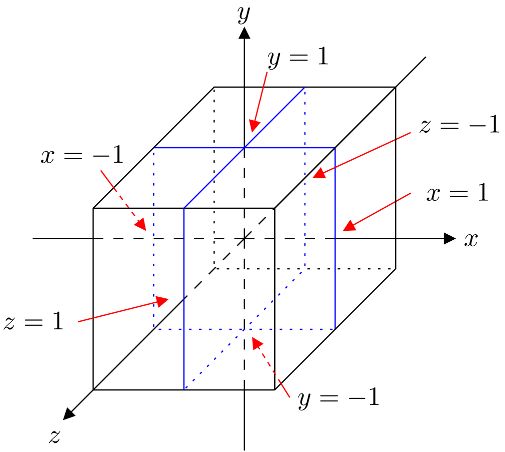
OpenGL Axes

- The positive direction of the z-axis follows the right-hand rule:



  - Pick two fingers and align one of them with the positive direction of x-axis and the other with the positive direction of y-axis
  - The positive direction of z-axis is the third finger

- OpenGL Canonical View Volume is a cube where
  - $x = -1.0$ to $x = 1.0$,
  - $y = -1.0$ to $y = 1.0$, and
  - $z = -1.0$ to $z = 1.0$.



OpenGL Canonical View Volume

- We can only see fragments/objects inside the OpenGL canonical view volume
- Viewing direction
  - Imagine that you are at the origin $(0.0, 0.0, 0.0)$
  - You look into the negative direction of z-axis
  - The top of your head is in the same direction as the positive direction of the y-axis
  - You will see every fragment that lies in between $x = \pm 1.0$, $y = \pm 1.0$, and $z = \pm 1.0$
- So for now, we are going to draw an object inside the view volume.

Creating a 3D Object

- An object comprises of a number of geometric primitives
  - points,
  - lines, and
  - polygons.
- A geometric primitive consists of one or more vertices
- A vertex is simply a point in a three-dimensional plane
- A point (x, y, z) is generally represented by a column vector with three elements:

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- However, we will actually represent a point by a column vector with four elements where the last element is always 1.0

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1.0 \end{bmatrix}$$

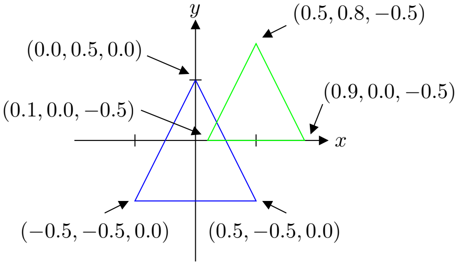
Creating a 3D Object

- **Note** that implementations of a vector in these slides is based on C `struct`. You can implement a vector as an array of floating-point as well.
- A column vector in OpenGL is a structure with four components:

```
typedef struct {
    GLfloat x;
    GLfloat y;
    GLfloat z;
    GLfloat w;
} vec4;
```

A Simple Triangle

- Our simple triangle lies on the plane $z = 0$ as shown below:



- The above triangle consists of three vertices, $(0.5, 0.0, 0.0)$, $(-0.5, -0.5, 0.0)$, and $(0.5, -0.5, 0.0)$ which can be defined in C as follows:

```
vec4 vertices[6] =
{{ 0.0,  0.5,  0.0, 1.0}, // top
 {-0.5, -0.5,  0.0, 1.0}, // bottom left
 { 0.5, -0.5,  0.0, 1.0}, // bottom right
 { 0.5,  0.8, -0.5, 1.0}, // top
 { 0.9,  0.0, -0.5, 1.0}, // bottom right
 { 0.1,  0.0, -0.5, 1.0}}; // bottom left
```

Front and Back

- Our triangle is a plane, one size is the front size and the other is the back
- OpenGL use right-hand rule to identify which side is front

- If the order of vertices are in the same direction of four fingers, the thumb indicates the direction of the front side.
- Note that from our example, the order of two triangles are different

- We will use RGBA color mode for this class
  - R, G, and B are the red, green, and blue elements and their value can be between 0.0 and 1.0
  - A is the alpha element (opacity) between 0.0 (fully transparent) and 1.0 (fully opaque)
- A color (R, G, B, A) is generally represented by a column vector with four elements:

$$\mathbf{c} = \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

Colors

- For this example, the first triangle is a multi-color triangle and the second triangle is simply green triangle
- Color of each vertex can be defined using the following code:

```
vec4 colors[6] =
{{1.0, 0.0, 0.0, 1.0},    // red   (for top)
 {0.0, 1.0, 0.0, 1.0},    // green (for bottom left)
 {0.0, 0.0, 1.0, 1.0},    // blue  (for bottom right)
 {0.0, 1.0, 0.0, 1.0},    // green (for top)
 {0.0, 1.0, 0.0, 1.0},    // green (for bottom right)
 {0.0, 1.0, 0.0, 1.0}};   // green (for bottom left)
```

The init() Function

- The `init()` function of our simple triangles is shown below:

```
void init(void)
{
    GLuint program = initShader("vshader.glsl", "fshader.glsl");
    glUseProgram(program);

    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    GLuint buffer;
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices) + sizeof(colors), NULL, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(colors), colors);

    GLuint vPosition = glGetAttribLocation(program, "vPosition");
    glEnableVertexAttribArray(vPosition);
    glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

    GLuint vColor = glGetAttribLocation(program, "vColor");
    glEnableVertexAttribArray(vColor);
    glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid *) sizeof(vertices));

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glDepthRange(1,0);
}
```

Load Program to Vertex and Fragment Shaders

- ```
  GLuint program = initShader("vshader.glsl", "fshader.glsl");
  ```
  Initialize shader programs
- ```
  glUseProgram(program);
  ```
  Installs a program object as part of current rendering state

Generate Vertex Array Object Name

- We are going to store a series of vertex attributes (locations, colors, etc) into the graphic pipeline
- A program may need multiple sets of series of vertex attributes (e.g., one for each object)
- We can also select which series of vertex attributes to render
- Use the following code:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

- `glGenVertexArrays()` generates vertex array object names
    - The first argument is the number of vertex array object names to generate
    - The second argument specifies the location of vertex array object names to be stored
- `glBindVertexArray()` binds a vertex array object for buffering data and rendering

Generate Vertex Array Object Name

- Multiple vertex arrays are suitable for drawing more than one object with different behavior
  - Body of a car moves (one vertex array)
  - Wheels of a car spin and move (another vertex array)
- This can be done by creating two vertex array objects

```
GLuint vao[2];
glGenVertexArray(2, &vao);

glBindVertexArray(vao[0]);
// Buffer data for the body of a car

glBindVertexArray(vao[1]);
// Buffer data for wheels
```

Transfer Data to the Graphic Pipeline

- Now we need to transfer our vertex attributes into the graphic pipeline
  - Need to allocate memory to store vertex attributes
  - Need to transfer data

```
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices) + sizeof(colors), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(colors), colors);
```

- The first three lines generates names object for buffer
- The fourth line allocate space where
  - Number of bytes is `sizeof(vertices) + sizeof(colors)`
  - `STATIC`: The data store contents will be modified once and used many times.
  - `DRAW`: The data store contents are modified by the application, and used as the source for GL drawing and image specification commands.

Transfer Data to the Graphic Pipeline

- `glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);`
  - Copy data from `vertices` to the currently binded buffer
  - The starting offset of the bind buffer is 0
  - The number of bytes to copy is `sizeof(vertices)`
- `glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(colors), colors);`
  - Copy data from `colors` to the currently binded buffer
  - The starting offset of the bind buffer is `sizeof(vertices)` (immediately after vertices data)
  - The number of bytes to copy is `sizeof(colors)`

Specify Data for Attributes in Vertex Shader

- ```
  GLuint vPosition = glGetAttribLocation(program, "vPosition");
  ```
  Locate the attribute named vPosition in the vertex shader program
- ```
  glEnableVertexAttribArray(vPosition);
  ```
  Enable the attribute vPosition
- ```
  glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
  ```
  Assign pointer to vPosition where arguments (from left to right) are as follows:
  - Specifies the index of the generic vertex attribute to be modified
  - Specifies the number of components per generic vertex attribute (vertices is vec4)
  - Specifies the data type of each component in the array (each component of vertices is GL_FLOAT)
  - specifies whether fixed-point data values should be normalized
  - Specifies the byte offset between consecutive generic vertex attributes (0 for tightly packed like struct)
  - Specifies a offset of the first component of the first generic vertex attribute in the array
  ```

Specify Data for Attributes in Vertex Shader

- Similarly for `vColor` (for `colors`) we use

```
GLuint vColor = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid *) sizeof(vertices));
```

- Note that `colors` has type `vec4` (four elements per attribute)
- The location locates immediately after `vertices`

Miscellaneous

- `glEnable(GL_DEPTH_TEST);`
  Enable hidden surface removal
- `glClearColor(0.0, 0.0, 0.0, 1.0);`
  Set the clearing color to black
- `glDepthRange(1,0);`
  Set depth range to 1.0 to 0.0 (some machine may not need this)

The display() function

- Lastly, the display function is as follows:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPolygonMode(GL_FRONT, GL_FILL);
    glPolygonMode(GL_BACK, GL_LINE);
    glDrawArrays(GL_TRIANGLES, 0, num_vertices);

    glutSwapBuffers();
}
```

  - Clear the color buffer and depth buffer
  - Fill polygon with color on the front side
  - Just lines on the back side
  - Draw triangles where the first vertex is located at offset 0 and the number of vertices is `num_vertices`
  - Swap drawing buffer to displaying buffert

Multiple Vertex Array Objects

- Suppose we want to draw a square and a triangle:

```
vec4 sqVertices[6] =
{{  0.5,   0.5,   0.0, 1.0}, // Top right
 { -0.5,   0.5,   0.0, 1.0}, // Top left
 { -0.5, -0.5,   0.0, 1.0}, // Bottom left
 {  0.5,   0.5,   0.0, 1.0}, // Top right
 { -0.5, -0.5,   0.0, 1.0}, // Bottom left
 {  0.5, -0.5,   0.0, 1.0}}; // Bottom right

vec4 sqColors[6] =
{{1.0, 0.0, 0.0, 1.0}, // Red
 {0.0, 1.0, 0.0, 1.0}, // Green
 {0.0, 0.0, 1.0, 1.0}, // Blue
 {1.0, 0.0, 0.0, 1.0}, // Red
 {0.0, 0.0, 1.0, 1.0}, // Blue
 {0.0, 1.0, 1.0, 1.0}}; // Aqua

vec4 triVertices[3] =
{{ 0.5,  0.8, -0.5, 1.0}, // top
 { 0.9,  0.0, -0.5, 1.0}, // bottom right
 { 0.1,  0.0, -0.5, 1.0}}; // bottom left

vec4 triColors[6] =
{{0.0, 1.0, 0.0, 1.0}, // blue
 {0.0, 1.0, 0.0, 1.0}, // blue
 {0.0, 1.0, 0.0, 1.0}}; // blue
```

Multiple Vertex Array Objects

- First generate two vertex array object names:

```
enum {square, triangle, numVAOs};
GLuint vao[numVAOs];
:
void init(void)
{
    GLuint program = initShader("vshader.glsl", "fshader.glsl");
    glUseProgram(program);

    GLuint vPosition = glGetAttribLocation(program, "vPosition");

    GLuint vColor = glGetAttribLocation(program, "vColor");

    glGenVertexArrays(numVAOs, vao);
    :
```

- We need the `vao` as a global variable since it will be used again for rendering
- `square`, `triangle`, and `numVAOs` are 0, 1, and 2, respectively.

Multiple Vertex Array Objects

- Buffer data for the square:

```
GLuint sqBuffer;

glBindVertexArray(vao[square]);
glGenBuffers(1, &sqBuffer);
glBindBuffer(GL_ARRAY_BUFFER, sqBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(sqVertices) + sizeof(sqColors), NULL,
                                                                     GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(sqVertices), sqVertices);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(sqVertices), sizeof(sqColors), sqColors);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid *) sizeof(sqVertices));
```

Multiple Vertex Array Objects

- Buffer data for the triangle:

```
GLuint triBuffer;

glBindVertexArray(vao[triangle]);
glGenBuffers(1, &triBuffer);
glBindBuffer(GL_ARRAY_BUFFER, triBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(triVertices) + sizeof(triColors), NULL,
                                                                  GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triVertices), triVertices);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(triVertices), sizeof(triColors), triColors);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid *) sizeof(triVertices));
```

Multiple Vertex Array Objects

- Rendering:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPolygonMode(GL_FRONT, GL_FILL);
    glPolygonMode(GL_BACK, GL_LINE);

    glBindVertexArray(vao[square]);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    glBindVertexArray(vao[triangle]);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glutSwapBuffers();
}
```

Helper Functions

- It is a good idea to implement helper function such as vector addition and scalar multiplication
- Example: vector addition:

```
vec4 v4v4_addition(vec4 v1, vec4 v2)
{
    vec4 result;

    result.x = v1.x + v2.x;
    result.y = v1.y + v2.y;
    result.z = v1.z + v2.z;
    result.w = v1.w + v2.w;

    return result;
}
```

- NOTES
  - The variable `result` is a local variable
  - A local variable of a function is located on the stack
  - Once the life-time of the function ends, data on the stack may be overwritten
  - Luckily, in C, a structure is passed by value

Helper Functions

- From previous code, we can perform the following:

```
vec4 v1 = {1.2, 2.3, 3.4, 0.0};
vec4 v2 = {-0.5, 4.9, -4.6, 0.0};
vec4 v3 = v4v4_addition(p1, p2);
```

- Note that since structure is passed by value:
  - Eight values must be passed when the function is called
  - Four values must be passed when the function returns
- The program may slow down if there are a large number of calculation (but not by much)

Helper Functions

- To increase the performance, consider pass by reference:

```
void v4v4_addition(vec4 *v1, vec4 *v2, vec4 *result)
{
    result->x = v1->x + v2->x;
    result->y = v1->y + v2->y;
    result->z = v1->z + v2->z;
    result->w = v1->w + v2->w;
}
```

- To use the function:

```
    vec4 v1 = {1.2, 2.3, 3.4, 0.0};
    vec4 v2 = {-0.5, 4.9, -4.6, 0.0};
    vec4 v3;
    v4v4_addition(&v1, &v2, &v3);
```

- Only three values are passed
- **Disadvantage**: make programming a little bit more complicate if you are not familiar with C pointers
- Use this strategy if you implement a vector as four-element array of floating-points

Implement Your Own Library

- Implement your own library (`.h` and `.c`)
  - Four-element column vector (`vec4`)
  - Scalar-vector multiplication and vector-vector addition functions
  - 4×4 (`mat4`) matrices (**COLUMN MAJOR** is preferred)
  - A 4×4 matrix is a row matrix with four column vector as shown below:

$$\begin{bmatrix}\mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3\end{bmatrix} \rightsquigarrow \begin{bmatrix}\begin{bmatrix}x_0\\y_0\\z_0\\w_0\end{bmatrix} & \begin{bmatrix}x_1\\y_1\\z_1\\w_1\end{bmatrix} & \begin{bmatrix}x_2\\y_2\\z_2\\w_2\end{bmatrix} & \begin{bmatrix}x_3\\y_3\\z_3\\w_3\end{bmatrix}\end{bmatrix}$$

- Scalar-matrix multiplication, matrix-matrix addition, and matrix-matrix multiplication functions
- Dot product, cross product, transpose, determinant, and reverse functions
- It is a good idea to have a couple of functions to print a vector or a matrix for debugging purpose