Computer Graphics

P. Healy

CS1-08
Computer Science Bldg.
tel: 202727
patrick.healy@ul.ie

Spring 2021–2022

Outline

Outline

1. Announcements
   - Labs (again)
2. OpenGL
   - Spinning Square
   - Other useful programs
3. Review of Matrices
   - Introduction
   - Arithmetic Operations
   - Transformations

Labs

- Week02 labs submitted on or before 09.00, Thu., Week03

Outline

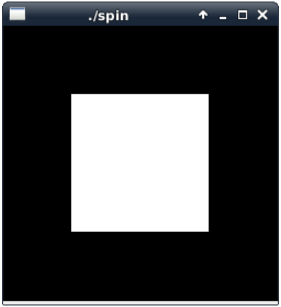
Animation requires 2 Buffers

- We'll look at an example of using the **mouse** to control spinning behaviour and using double buffering to properly render changing displays
- Program initially draws white square on the screen (at rest)
- When the left mouse is pressed down it begins spinning
- When the middle or right mouse is pressed it stops
- In `display()` function the rotation is around the *z*-axis, the vector , which comes out of page
- Code that follows is ordered for *presentation* purposes

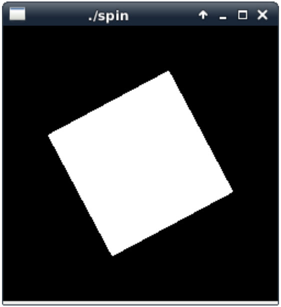
main()

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape); // for screen resizing
   glutMouseFunc(mouse);  //  what's this?
   glutMainLoop();
   return 0;   /* ANSI C requires main to return in
}
```

`spin` at rest



`spin` at play



Mouse Interaction

```
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay); //

         break;
      case GLUT_MIDDLE_BUTTON:
      case GLUT_RIGHT_BUTTON:
         if (state == GLUT_DOWN) glutIdleFunc(NULL)
         break;
      default:
         break;
   }
```

display()

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix(); // what's this?
   glRotatef(spin, 0.0, 0.0, 1.0); // z-axis rotati
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();

   glutSwapBuffers();
}
```

Our Initialisation

```
#include <GL/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT); // what's this?
}
void spinDisplay(void)
{
   spin = spin + 2.0;
   if (spin > 360.0)  spin = spin - 360.0;
   glutPostRedisplay();  // what's this?
}
```

Outline

Cube example

- An example from “Redbook” of an OpenGL generated perspective on a cube
- “Redbook” examples directory has lots of useful toy programs to demonstrate different features

Cube example

- An example from “Redbook” of an OpenGL generated perspective on a cube
- “Redbook” examples directory has lots of useful toy programs to demonstrate different features

Outline

What is a matrix?

Good review of vectors and matrices here

A **matrix** is a rectangular array of numbers:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

A is an $m \times n$ matrix and the element is the element at row i and column j.

Two matrices A and B are identical if they are of the same dimension and $a_{ij} = b_{ij}, \forall 1 \leq i \leq m, \text{and} 1 \leq j \leq n$.

We can **add**, **subtract** and **multiply** matrices; “division” can only be achieved by multiplying by a matrix’s **inverse**.

Some important matrices

The identity matrix, I, is a square matrix, $n \times n$, with 1s on the main diagonal and 0s everywhere else

$$I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \cdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Matrix of all 1s, sometimes called J, or $\mathbf{1}$; matrix of all 0s, $\mathbf{0}$.

A diagonal matrix, $D = (d_1, d_2, \ldots, d_n)$, is a square matrix, $n \times n$, with d_is on the main diagonal and 0s everywhere else

$$D = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \cdots & d_n \end{pmatrix}$$

Outline

Addition

We can only add matrices of identical size

$$C = A + B$$

only makes sense if A, B, and C are all of dimension $m \times n$
$c_{ij} = a_{ij} + b_{ij}, \forall i, j$
Properties:

- Associativity: $(A + B) + C = A + (B + C)$
- Commutativity: $A + B = B + A$

Subtraction

We can only subtract matrices of identical size

$$C = A - B$$

only makes sense if A, B, and C are all of dimension $m \times n$
$c_{ij} = a_{ij} - b_{ij}, \forall i, j$
Properties:

- Associativity: $(A - B) - C \neq A - (B - C)$
- Commutativity: $A - B \neq B - A$

Multiplication

Scalar multiplication: given a single number (a scalar) we can "scale" every element of a matrix by this number:

$$C = kA$$

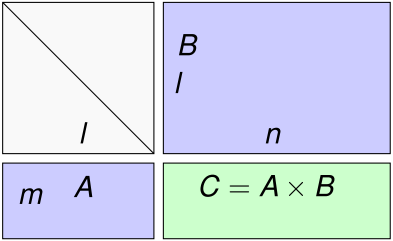
This means that $c_{ij} = ka_{ij}, \forall i, j$
Properties:

- Associativity: $(k_1 \times k_2)A = k_1 \times (k_2 \times A)$
- Commutativity: $k_1 A = Ak_1$
- Distributivity: $k(A + B) = kA + kB$

Multiplication (contd.)

Matrix multiplication: can only multiply two matrices A and B if no. of cols of A equals no. of rows of B

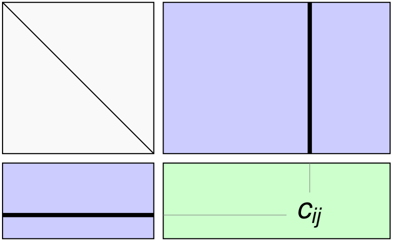
$$C_{mn} \leftarrow A_{ml} \times B_{ln}$$



Multiplication (contd.)

Matrix multiplication: can only multiply two matrices A and B if no. of cols of A equals no. of rows of B

$$c_{ij} = \sum_{k=1}^{l} a_{ik} b_{kj}$$



- Element c_{ij} is the **dot product** of row i of A and column j of B
- That is, the sum of the products of the kth elements of each, $1 \leq k \leq l$

Multiplication (contd.)

Since matrix multiplication relies crucially on col. count of left and row count of right, we may not be able to compute $C' = BA$ eventhough $C = AB$.

Properties:

- Associativity: $A(BC) = (AB)C$
- Commutativity: $AB \neq BA$
- Distributivity: $C(A + B) = CA + CB$

For identity matrix, I,

$$AI = IA$$

The identity matrix plays role of 1 in arithmetic.

Outline

1. Announcements
   - Labs (again)
2. OpenGL
   - Spinning Square
   - Other useful programs
3. Review of Matrices
   - Introduction
   - Arithmetic Operations
   - Transformations

Vectors

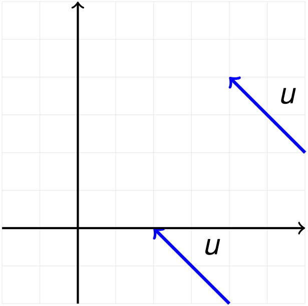
- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere



- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Vectors

- A **vector** is defined to be the *difference* between two points
- In 2-D the vector, u, from p to q is $u = (q_x - p_x, q_y - p_y)$
- This means that a vector has a *direction* and a *magnitude* (length) but it doesn't have a *position*: it can exist anywhere
- For convenience vectors are often drawn pointing from the origin; if we do this then a point, $p = (x, y)$, can be represented by the vector u from the origin to p, $u = (x - 0, y - 0) = (x, y)$
- In n-dimensional space we can collect the n components of a vector and write it in matrix form
- A $1 \times n$ matrix (1 row, n columns) is called a row vector
- An $n \times 1$ matrix (n rows, 1 column) is called a **vector** or a column vector

Transforming a point

- A core notion in graphics is the idea of moving points around the screen
- This can be because we want to **rotate** some object to get a different view on it, or it may be because we want to **scale** or **translate** (shift sideways) it
- A **transformation** maps each point, p, into a new point, q, using a specific formula (or algorithm)
- We can write this transformation as $T(p) \to q$
- As we will see here a matrix transformation can be used to map whole sets of points

Transforming a point

- A core notion in graphics is the idea of moving points around the screen
- This can be because we want to **rotate** some object to get a different view on it, or it may be because we want to **scale** or **translate** (shift sideways) it
- A **transformation** maps each point, p, into a new point, q, using a specific formula (or algorithm)
- We can write this transformation as $T(p) \to q$
- As we will see here a matrix transformation can be used to map whole sets of points

Transforming a point

- A core notion in graphics is the idea of moving points around the screen
- This can be because we want to **rotate** some object to get a different view on it, or it may be because we want to **scale** or **translate** (shift sideways) it
- A **transformation** maps each point, p, into a new point, q, using a specific formula (or algorithm)
- We can write this transformation as $T(p) \rightarrow q$
- As we will see here a matrix transformation can be used to map whole sets of points

Transforming a point

- A core notion in graphics is the idea of moving points around the screen
- This can be because we want to **rotate** some object to get a different view on it, or it may be because we want to **scale** or **translate** (shift sideways) it
- A **transformation** maps each point, p, into a new point, q, using a specific formula (or algorithm)
- We can write this transformation as $T(p) \rightarrow q$
- As we will see here a matrix transformation can be used to map whole sets of points

Transforming a point

- A core notion in graphics is the idea of moving points around the screen
- This can be because we want to **rotate** some object to get a different view on it, or it may be because we want to **scale** or **translate** (shift sideways) it
- A **transformation** maps each point, p, into a new point, q, using a specific formula (or algorithm)
- We can write this transformation as $T(p) \to q$
- As we will see here a matrix transformation can be used to map whole sets of points

Affine Transformation

- Of all the transformations used in graphics the **affine transformation** is one of the most common
- The affine transformation (AT) guarantees that points map to points and parallel lines map to parallel lines (whose separation may change!)
- Five main examples of AT: **translation**, **rotation**, **scaling**, **reflection** and **shear**