DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE

Moving Shapes & Player Input

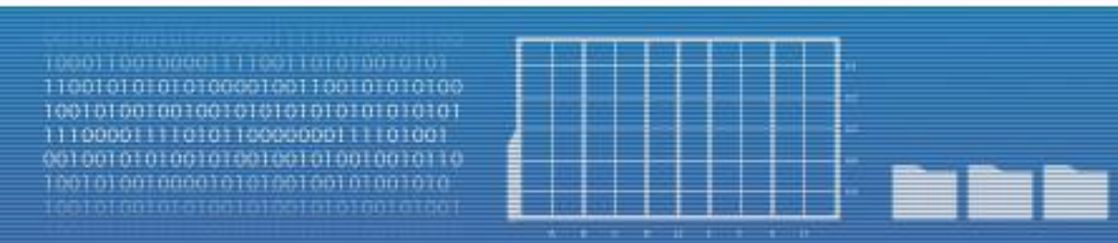
Movement and the math behind it

Lecture Time!

- Vectors: Math
- Circles: More Math
- SFML: Now With More Math
- Keyboard: Keys
- Mouse: Buttons and Position

WARNING

- This should be a review of linear algebra
- Since this subject is primarily for DGDD majors, I am going to fast-forward through a lot of this



Vectors

- An n-tuple of numbers from the domain of real numbers
  - Number of dimensions = n
  - Therefore, an example of a 2-D vector would be (4, –8.9)

Vectors

- Interpreted as displacement from the origin to a specific point
  - Position
  - Can be used for other things (like a rectangle's size)

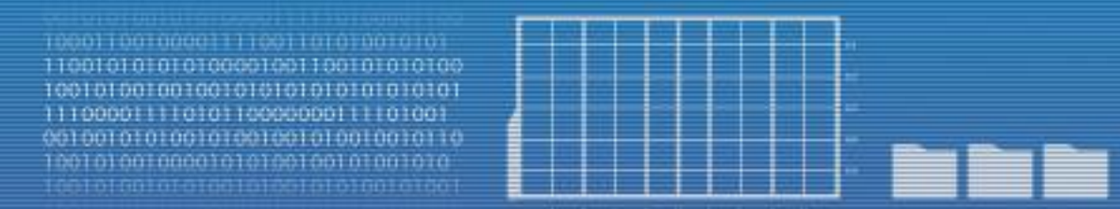
Vector Arithmetic

- Vector addition:
  - $a = (a_x , a_y)$, $b = (b_x , b_y)$
  - $a + b = ?$

- What is the resulting vector?
  - $a + b = (a_x + b_x, a_y + b_y)$

Vector Arithmetic

- Useful for:
  - Translation (motion, update of position)
  - Subtraction (a – b = a + (– b))



Vector Arithmetic

- Vector subtraction:
  - a = (a_x , a_y), b = (b_x , b_y)
  - a - b = ?

- What is the resulting vector?
  - a – b = (a_x – b_x, a_y – b_y)
  - What is the resulting vector if b is used as an origin?

Vector Arithmetic

- Useful for:
  - Distance (between two points, used in conjunction with getting a vector's magnitude)
  - Getting an "opposing" vector (faces the opposite direction, same magnitude)

Vector Arithmetic

- Multiplying a vector with a scalar:
  - s is a constant, $a = (a_x , a_y)$

- What is the resulting vector?
  - $sa = (sa_x, sa_y)$

Vector Arithmetic

- Useful for:
  - Scaling (shrinking or enlarging relative to a local origin)
  - Normalization
- Division is just multiplying by a reciprocal of the divisor

Vector Arithmetic

- Getting the magnitude of a vector:

  $|a| = \mathrm{sqrt}(a_x^2 + a_y^2)$

  - What is magnitude, assuming a line segment formed by point a and the origin?

Vector Arithmetic

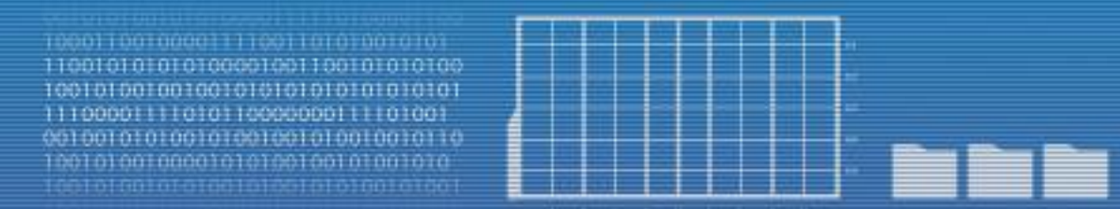
- Useful for:
  - Distance (between two points, used in conjunction with subtraction)
  - Speed (given a velocity vector)

Vector Arithmetic

- Normalizing a vector:

$$\hat{a} = (a_x / |a|, a_y / |a|)$$

  - What is the resulting vector?



DISCS

Vector Arithmetic

- Useful for:
  - Emphasis on direction without scale (using non-unit vectors may add extra “corrective” steps in some operations)

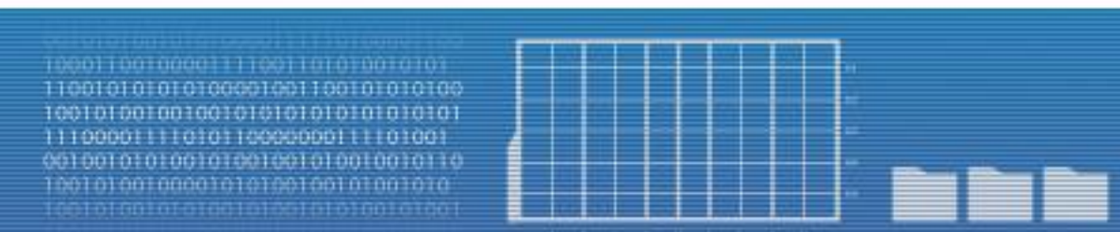
Dot Product

- Dot product of two vectors:

$$a \bullet b = a_x b_x + a_y b_y$$

  - Also known as scalar product

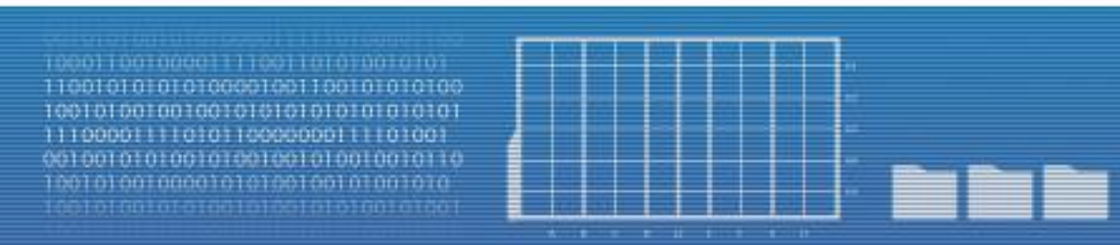
- Useful for:
  - Lots of stuff



DISCS

Dot Product

- $a \bullet b = |a||b| \cos \theta$
  - θ being the smallest angle between a and b
- $a \bullet a = |a|^2$

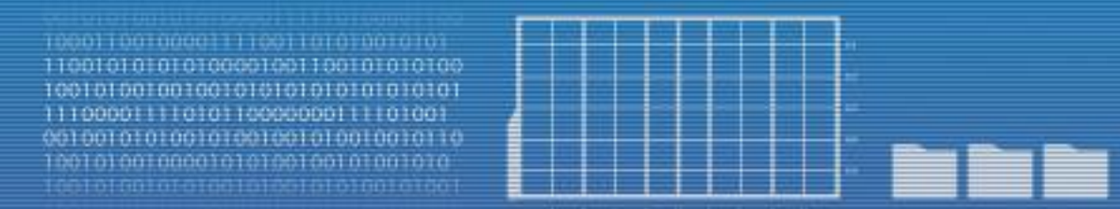


Pseudo Cross Product

- Perp-dot product of two vectors: $a^P \bullet b$
  - a^P is counterclockwise vector perpendicular to a
  - $a = (x, y)$, $a^P = (-y, x)$
- Useful for:
  - Determining if b is counterclockwise (positive), clockwise (negative), or along a (zero)

Lines

- A line can be defined as the set of points expressible as the linear combination of two distinct points A and B:
  - $L(t) = (1 - t)A + tB = A + t(B - A)$
    - In other words, just need a point and a direction
- A ray is a line but $t \geq 0$
- A segment is a line but $0 \leq t \leq 1$



Projection onto a Vector

- Useful for things like Separating Axis Theorem
- Assume u is a unit length vector and any other vector v
- The projection of v onto u is another vector along u
  - Can be expressed as some value L multiplied by u

Projection onto a Vector

- L = v ● u
- The projection of v onto u therefore is
  proj(v, u) = (v ● u)u
  - Proof will be or was already taught in Linear Algebra
- Projecting v onto a non-unit vector d:
  proj(v, d) = ((v ● d) / (d ● d))d

Circles

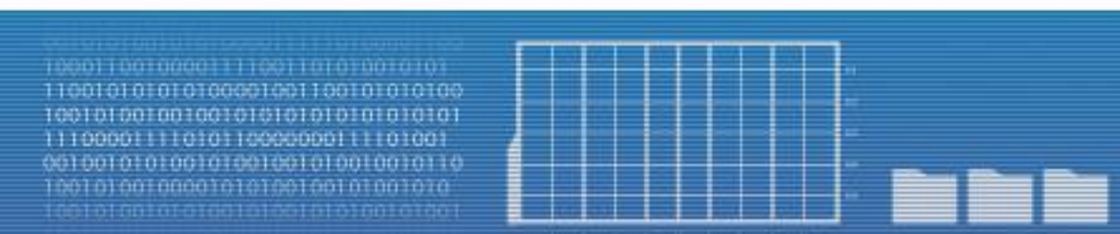
- Usually rendered as a polygon with many sides
- Represented by a vector indicating its center and a value representing its radius
- Most memory-efficient compared to other 2-D shape representations

Circles

- Overlap test involves checking if the distance between two circle centers is less than or equal to the sum of their radii
  - But square root operations are expensive
  - Is there a way to perform this check without the square root operation?

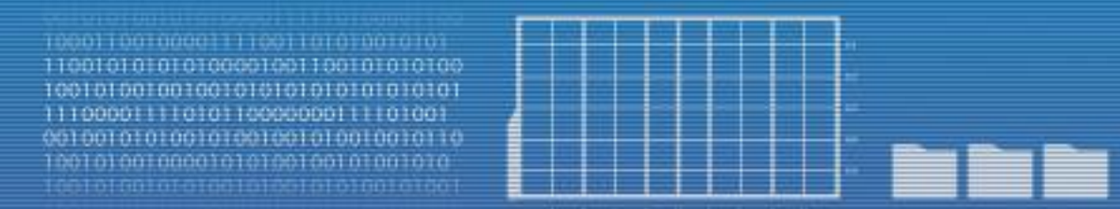
SFML Window

```
int main()
{
    // create window
    sf::RenderWindow window(
        sf::VideoMode( 480, 320 ),
        "Title Goes Here" );
// other initializations here
// ...
```



SFML Window

```
// ...
while( window.isOpen() )
{
	// check all the window's events that were triggered
	// since the last iteration of the loop
	sf::Event event;
	while( window.pollEvent( event ) )
	{
		// "close requested" event
		if( event.type == sf::Event::Closed )
		{
			window.close();
		}
	}
// ...
```



SFML Graphics

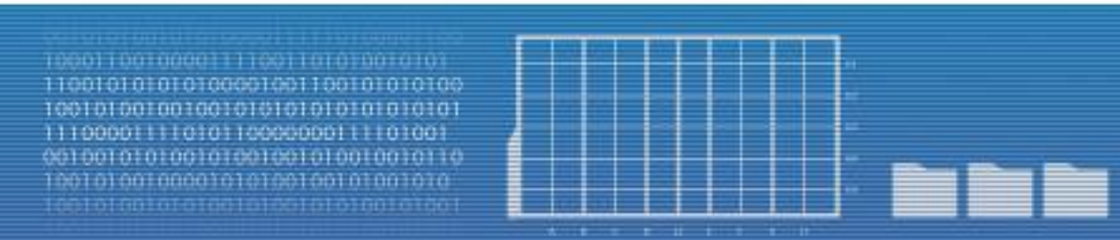
```
// in initialization part, after creating window
sf::CircleShape circ;
sf::RectangleShape rect;
circ.setPosition( 40, 40 );
circ.setRadius( 100.0f );
circ.setFillColor( sf::Color( 0, 255, 0 ) );
rect.setPosition( 340, 180 );
rect.setSize( sf::Vector2f( 100.0f, 60.0f ) );
rect.setFillColor( sf::Color( 0, 0, 255 ) );
```

SFML Graphics

```
// in window.isOpen() loop
// always clear buffer at start of current frame
window.clear( sf::Color::Black );

// draw shapes
window.draw( circ );
window.draw( rect );

// always call next line at end of current frame
window.display();
```

Exercises

- Make programs that draw the following:
  - RectangleShape that travels left and right
  - CircleShape that also moves in a circle
  - RectangleShape that slowly changes size and color
    - Should eventually loop back to its original size and color



Exercises

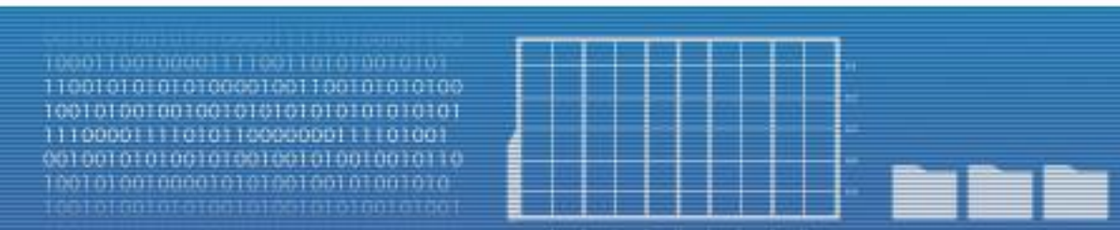
- Ensure that your programs run at (roughly) 60 frames per second
- Normally good practice to have the values be read from a file for testing
  - But as this is an exercise so there's no problem hard-coding the values here

SFML Keyboard Input

```
// optional initialization (for keybinding)
sf::Keyboard::Key keyUp = sf::Keyboard::W;
sf::Keyboard::Key keyDown = sf::Keyboard::S;
sf::Keyboard::Key keyLeft = sf::Keyboard::A;
sf::Keyboard::Key keyRight = sf::Keyboard::D;
sf::Keyboard::Key keyQuit = sf::Keyboard::Escape

// additional code / a workaround is required
// to make this work in a switch-case statement
```

SFML Keyboard Input

```
// in window.isOpen() loop
if( sf::Keyboard::isKeyPressed( keyQuit ) )
{
      window.close();
}
if( sf::Keyboard::isKeyPressed( keyUp ) )
{
      // ...
}
// ...
```



DISCS

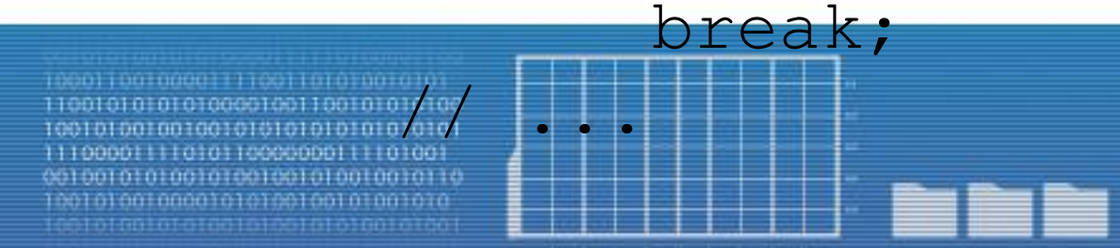
SFML Keyboard Input

- Note that the `isKeyPressed()` function will return `true` as long as the key is held down
- You will have to provide your own programming logic if you only want the "first" `true` return value to trigger something
  - Hint: Need another `bool`

SFML Keyboard Input

- You may also consider using events

```
while( window.pollEvent( event ) )
{
        switch( event.type )
        {
        // ...
        case sf::Event::KeyPressed:
              switch( event.key.code )
              {
              case sf::Keyboard::Escape:
                    window.close();
                    break;
              // more cases here for the other keys
              }
              break;
        //  ...
```



SFML Keyboard Input

```
// event loop, continued
        // ...
        case sf::Event::KeyReleased:
                // similar to sf::Event::KeyPressed
                break;
        // ...
```

SFML Keyboard Input

- Note that multiple `KeyPressed` events will be generated at a rate dependent on your OS settings if a key is held down
- Can be disabled in the initialization part of your code (force a maximum of 1 `KeyPressed` event until key is released)

```
// additional initialization
window.setKeyRepeatEnabled(false);
```

SFML Keyboard Input

- The event system is similar to Java's
  - In other words, don't put large blocks of code in it
  - You're much better off using flags
    - Flags = value that acts as a signal for the program to do something later on

SFML Keyboard Input

- For `KeyPressed` events, simply flag that the relevant key/s have been pressed (with a bit or a `bool` per key – set to true)
- For `KeyReleased` events, simply flag that the relevant key/s have been released (set to false)

SFML Keyboard Input

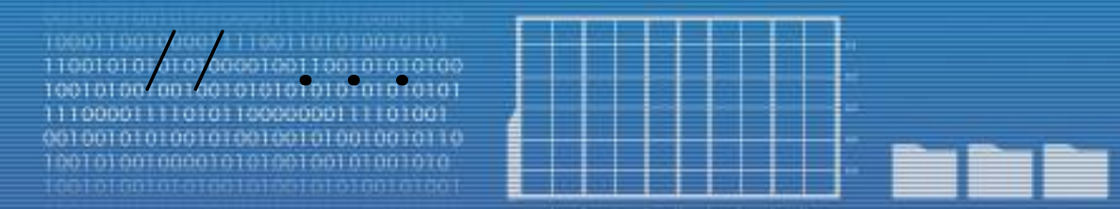
- Then, in the relevant AI part of your code, simply check those flags to see what needs to be run
  - While W/A/S/D flags are true, move the player character
  - Reset flag/s to false for keys that are meant to be pressed repeatedly and not held down

SFML Keyboard Input

- This can also apply for other things not related to immediate movement
  - If ESC is pressed, you can exit the program or open a menu (depending on your settings)
  - If SPACE is pressed, you can have your character jump or do some other action (depending on the setting)

SFML Mouse Input

```
// in window.isOpen() loop
// note: vector returned uses
// same coordinate system as Shape positions
// note#2: window can be omitted
// to get position relative to desktop instead

sf::Vector2i mPos =
      sf::Mouse::getPosition( window );

if( sf::Mouse::isButtonPressed(
      sf::Mouse::Left ) )

{

      // ...

}

// ...
```



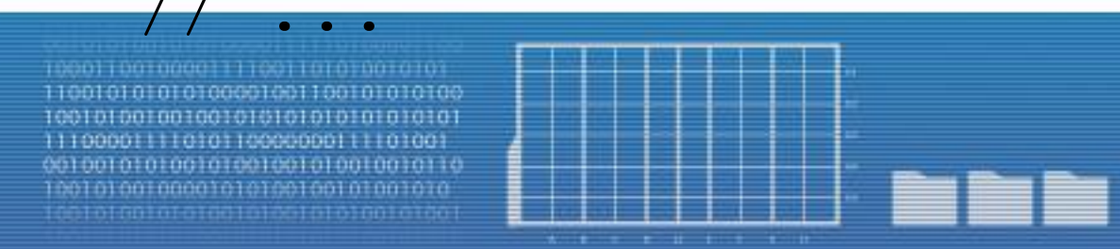
SFML Mouse Input

- Again, the `isButtonPressed()` function will return `true` as long as the mouse button is held down

- You may also consider using events
  - Mouse wheel input can only be handled using events

SFML Mouse Input

```
// additional cases to event loop's switch statement
case sf::Event::MouseButtonPressed:
        switch( event.mouseButton.button )
        {
        case sf::Mouse::Left:
                // set flag to true
                break;
        case sf::Mouse::Right:
                // set flag to true
                break;
        }
        break;
case sf::Event::MouseButtonReleased:
        // same, but set flags to false
        break;
// ...
```



SFML Mouse Input

```
// additional cases to event loop's switch statement
// ...
case sf::Event::MouseMoved:
      // set mouse delta values
      // based on mouse position
      // from previous frame
      break;
// ...
```

Exercise

- Draw 3 RectangleShapes, each with a different color (red, green, blue)
  - Clicking on one of them should "select" it and change its color to white
  - This should also deselect any RectangleShape that was selected earlier and revert it to its original color
  - Clicking on an empty space should also cause this deselect action

Exercise

- Use the keyboard (WASD keys) to move the selected RectangleShape

- To ease worries about overlaps, it's okay for it to only select one of the three RectangleShapes if you click on a space shared by all of them
  - Make sure only one gets selected, though!

Homework

- Normally, the homework would be a two-part homework
  - Due to the change from semester to quarter and the resulting loss of time, the two homework were pruned and combined

Homework

- Create an array of 60 CircleShapes and 40 RectangleShapes
  - CircleShape radius is 30.0f
  - RectangleShape dimensions are 50.0f x 50.0f
- Assign different colors to each shape
  - Cycle through red, green, blue, yellow, cyan, and white
  - Do consider a convenience function for this

Homework

- CircleShapes should drift downwards at a rate of 20 pixels per second
- RectangleShapes should drift to the right at a rate of 20 pixels per second
- Not 20 pixels per tick (movement should be very smooth)

Homework

- Starting positions of these shapes should be random
  - (0, 0) to (window_w – 1, window_h – 1)

- These values should be stored in settings.txt so we can easily change them for testing purposes

Homework

- The first element in your CircleShape array should not drift downwards
- It should respond to WASD keys and move in the corresponding direction at a rate of 200 pixels per second

Homework

- Speed should be adjusted accordingly if diagonal movement is detected
  - It should cover 200 pixels per second no matter what direction it is moving
- Older games had some issues where moving diagonally moved the entity more
  - It moved horizontally *and* vertically for the same distance at the same time
  - This leads to a higher distance travelled

Homework

- The first element in your RectangleShape array should no longer drift to the right
- It should now move directly towards the cursor at a rate of 200 pixels per second while the left mouse button is held down
- Again, speed should be adjusted accordingly if movement is along both horizontal and vertical axes

Homework

- BONUS
  - Fix the jittering in the uploaded sample
    - This has to do with the precision of the mouse coordinates and the shape trying to adjust accordingly in real time
  - Not something necessary for full points but a very good thing to practice since you don’t want this jittering in your real games