CS230
Game Implementation Techniques

Lecture 9

Overview

- Object Kinematics
- Object Movement
  - Frame Based
  - Time Based
  - Acceleration
- Asteroids
  - Ship's Acceleration, Deceleration
  - Velocity Cap
  - Friction

Object Kinematics (1/2)

- An object has a position and a velocity
  - Objects do respond to forces – We won't simulate this for now.
  - If objects move with constant velocity – that is, zero acceleration, it is the simplest to simulate.

Object Kinematics (2/2)

- Obvious structure definition in C might look like (neglecting appearance and other properties):

```
struct Object
{
        … // Object methods and variables
        float  p[2];    // Position
        float  v[2];    // Velocity
};
```

Object Kinematics (2/2)

- Same as:

```
struct Object
{
        … // Object methods and variables
        AEVec2  p;     // Position
        AEVec2  v;     // Velocity
};
```

Overview

- Object Kinematics
- Object Movement
  - Frame Based
  - Time Based
  - Acceleration
- Asteroids
  - Ship’s Acceleration, Deceleration
  - Velocity Cap
  - Friction

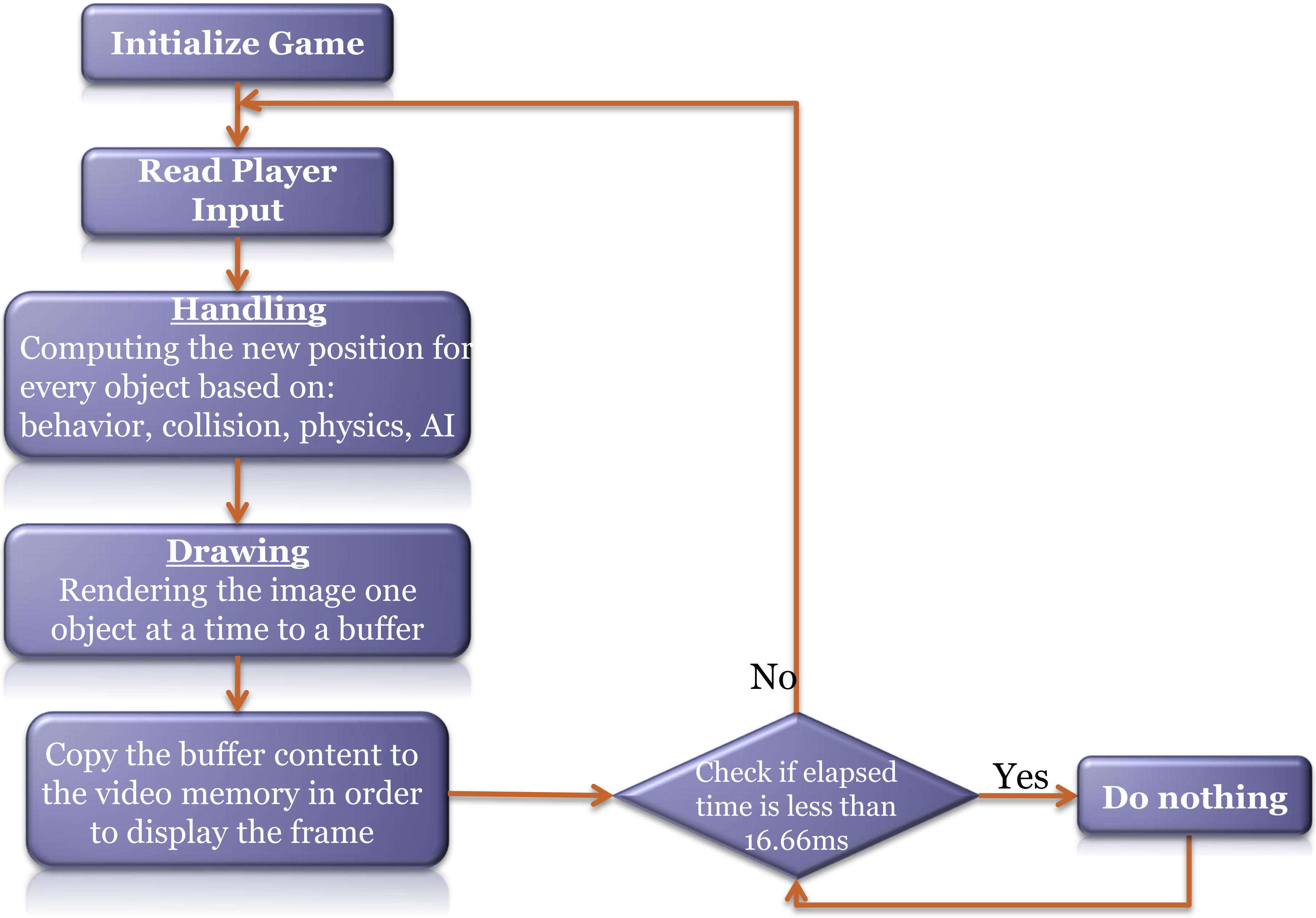
Object Movement

- Specify the initial position *p* and velocity *v* of each object
  - Velocity consists of a speed and a unit direction vector (that is, a vector with unit magnitude)

- Every frame, update object's previous position:

$$\vec{\boldsymbol{p}} += \vec{\boldsymbol{v}}$$

- This movement type is called "Frame Based"


Initialize Game
Read Player Input
Handling
Computing the new position for every object based on: behavior, collision, physics, AI
Drawing
Rendering the image one object at a time to a buffer
Copy the buffer content to the video memory in order to display the frame
Check if elapsed time is less than 16.66ms
No
Yes
Do nothing

Better Game Loop (1/2) - Revisited

- Objects are no longer updated based on a pre-determined time between successive frames
- Instead, time interval to complete current frame is used in kinematics calculations to determine objects' displacements
  - Computing time interval to complete current frame is non-trivial problem
  - Instead, good compromise is to use time interval of previous frame

Better Game Loop (2/2) - Revisited

```
double t = 0.0f; // game time (in seconds)
double currTime = time( ); // measure time at start of frame
Initialize_Game_Objects( t, 0.0f );
Draw_Game_Objects( );

while (!quit)
{
   double newTime = time( ); // measure time at end of previous frame or time at
   start of current frame
   double dt = newTime - currTime; // time interval for previous frame (in seconds)
   currTime = newTime; // time at start of current frame
   Update_Game_Objects( t, dt );
   Draw_Game_Objects( );

   // Lock the frame rate here - Waste time (NOT A REQUIRED STEP ANYMORE)

   t += dt; // update game time with time interval of previous frame
}
```

Object Movement (Revisited) (1/6)

- Specify the initial position *p* and velocity *v* of each object
  - Velocity consists of a speed and direction vector (that is, a vector with unit magnitude)

Object Movement (Revisited) (2/6)

- Each frame:
  - Compute time interval between previous and current frame: dt
  - Compute object's displacement within time interval *dt:* $\vec{v} * dt$
  - Finally, compute object's new position as

$$\boldsymbol{\vec{p} += \vec{v} * dt}$$

- This movement type is called "Time Based"

Object Movement (Revisited) (3/6)

- Example: Frame based
- Velocity is: $\vec{v} = (3, 0)$

$$\vec{p} += \vec{v}$$

- Along the x-axis:
  - At 60 FPS, the object will move 180 units per second
  - At 30 FPS, the object will move 90 units per second
  - At X FPS, the object will move 3*X units per second

Object Movement (Revisited) (4/6)

- Example: Time based
- Velocity is: $\vec{v} = (180, 0)$ $\vec{p} += \vec{v} * dt$

- Along the x-axis:
  - At 60 FPS, the object will move ? units per second
  - At 30 FPS, the object will move ? units per second
  - At X FPS, the object will move ? units per second
    - Assuming X is not equal to 0

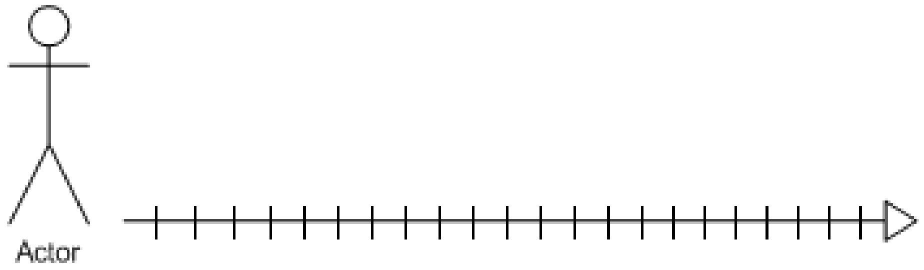
Object Movement (Revisited) (4/6)

- Example: Time based
- Velocity is: $\vec{v} = (180, 0)$ $\vec{p} += \vec{v} * dt$

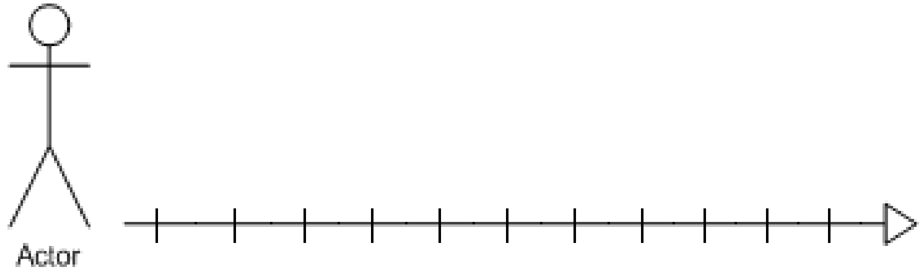
- Along the x-axis:
  - At 60 FPS, the object will move 180 units per second
  - At 30 FPS, the object will move 180 units per second
  - At X FPS, the object will move 180 units per second
    - Assuming X is not equal to 0

Object Movement (Revisited) (5/6)

- In time based games, the step size will adjust according to the frame time
- 60 FPS:



- 30 FPS:



Object Movement (Revisited) (6/6)

- Conclusion:

  In a time based application, given a time period, a linearly animated object will always reach the same position, independently from the game's frame rate. What differs is the smoothness of the movement, where a slow FPS will make the character look as if it's disappearing and reappearing at its new location (Which is technically true!) instead of creating the illusion of motion.

Object Movement – Based on Velocity

- Compute time interval between previous and current frame
  - dt
- Compute object's displacement within time interval dt:
  - $v * dt$
- Finally, compute object's new position as

$$newPos = v * dt + currPos$$

Object Movement – Based on Acceleration

- Computing object's new position as:

$$newPos = \frac{1}{2} a * dt^2 + v * dt + currPos$$

References

- Computer Graphics Principles and Practice by Foley, van Dam, Feiner and Hughes

Overview

- Object Kinematics
- Object Movement
  - Frame Based
  - Time Based
  - Acceleration
- Asteroids
  - Ship’s Acceleration, Deceleration
  - Velocity Cap
  - Friction

Asteroids

- Bullets & Asteroids have constant velocities
    - Velocities are set at creation time
- The ship has a varying velocity
    - Depending on its acceleration, which in turn depends on user input
    - Its acceleration is non-zero when either the forward or backward key is pressed

Asteroids – Ship's Acceleration (1/5)

- The ship's new position can be calculated in 2 ways
  - Directly from the acceleration:

$$newPos = \frac{1}{2} a * dt^2 + currVel * dt + currPos$$

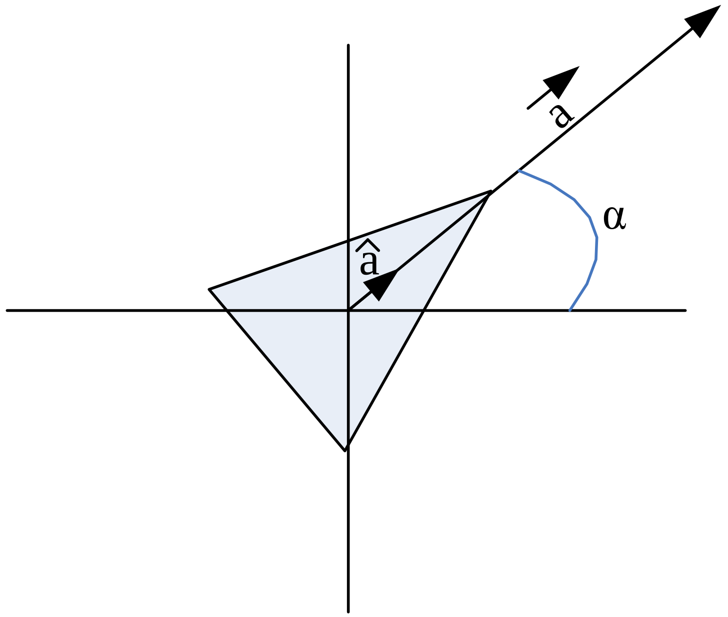
  - Calculate the new velocity, then use it to get the new position:

$$newVel = a * dt + currVel$$

$$newPos = newVel * dt + currPos$$

Asteroids - Ship's Acceleration (2/5)

- When the forward button is pressed, a forward acceleration should be applied to the ship
- Ship data that we have:
  - Ship's current position
  - Ship's current velocity
  - Ship's current orientation: α
- What we need to calculate:
  - Ship's acceleration: $\vec{a}$



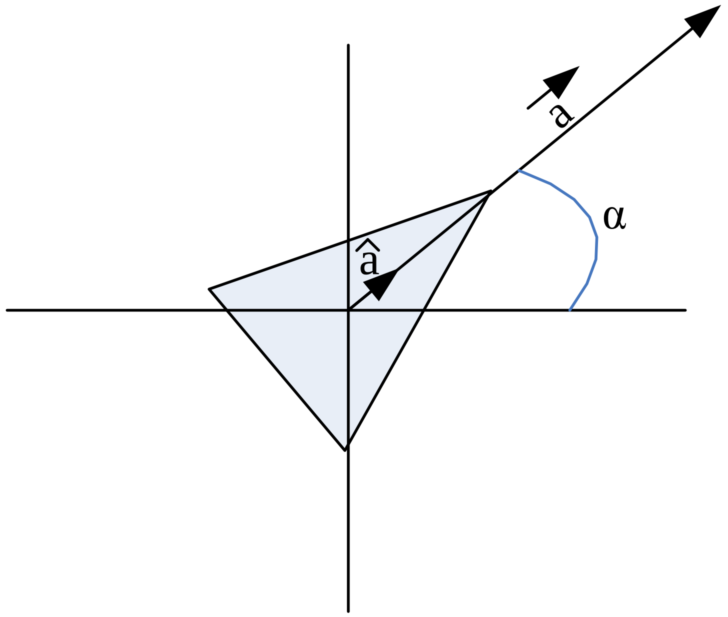
Asteroids – Ship's Acceleration (3/5)

- **The new acceleration vector** $\vec{a}$ **is independent from its current velocity**
- We can use the ship's current orientation α to compute the normalized acceleration vector: $\hat{a}$

$$\hat{a} = (\cos\alpha\,;\sin\alpha)$$

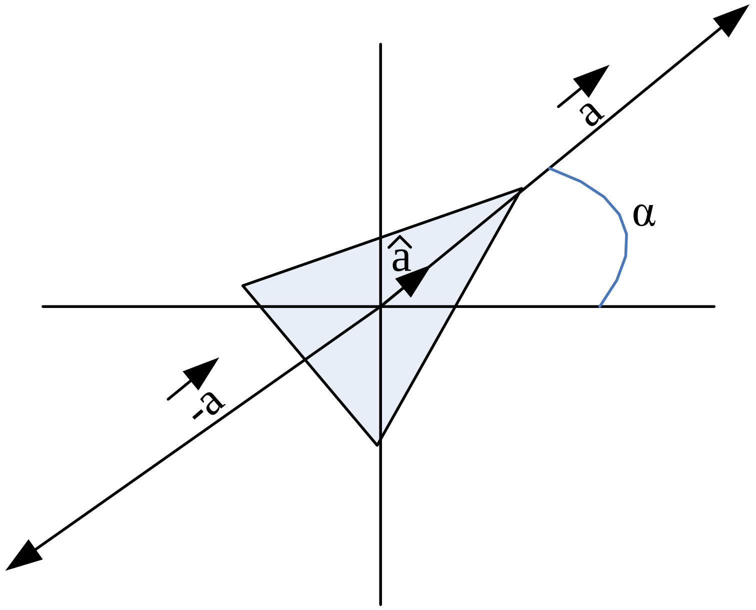
- Scaling $\hat{a}$ by a predefined value will give the full acceleration vector $\vec{a}$

$$\vec{a} = (\hat{a}.x * 100; \hat{a}.y * 100)$$



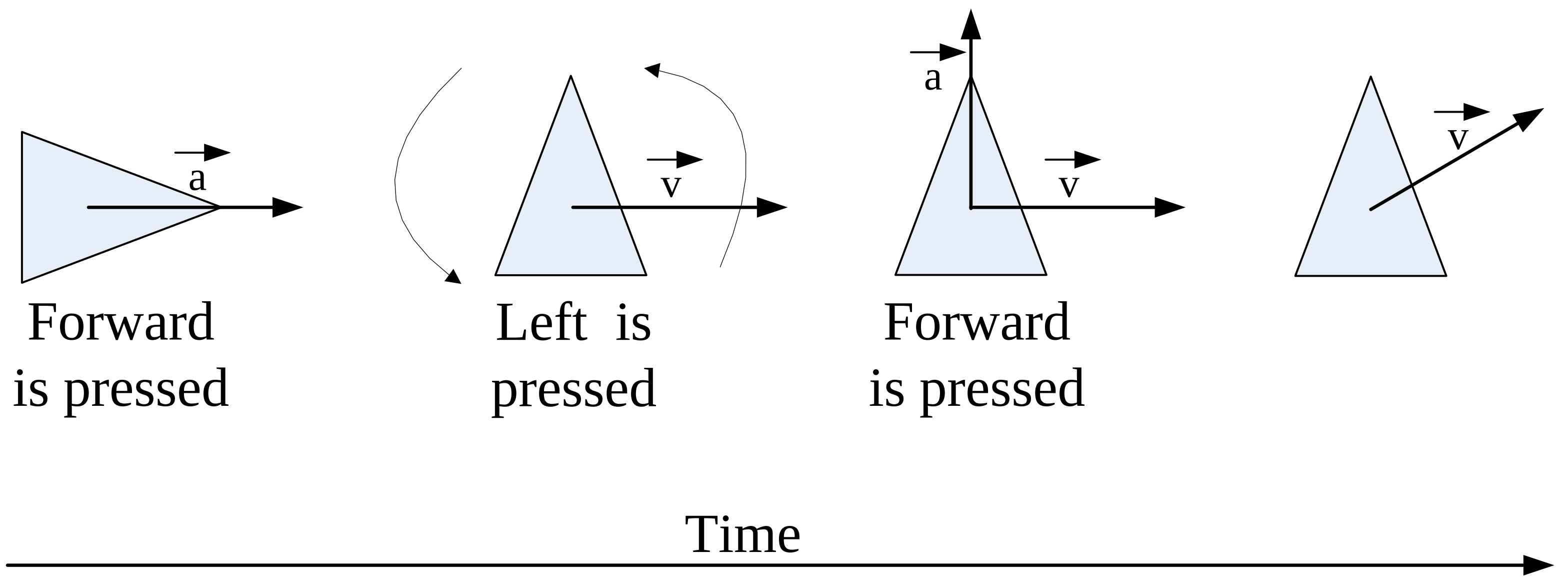
Asteroids - Ship's Acceleration (4/5)

- The deceleration vector $-\vec{a}$ is similarly calculated
  - It's just the opposite vector
- Compute $\vec{a}$ as described previously
- Negate both coordinates to get $-\vec{a}$



Asteroids – Ship's Acceleration (5/5)

- Assuming the ship is initially not moving



Asteroids – Ship’s Velocity Cap (1/4)

- Both techniques will achieve slightly different results, because of different approximations

$$newVel = a * dt + currVel$$
$$newPos = \frac{1}{2} a * dt^2 + currVel * dt + currPos$$

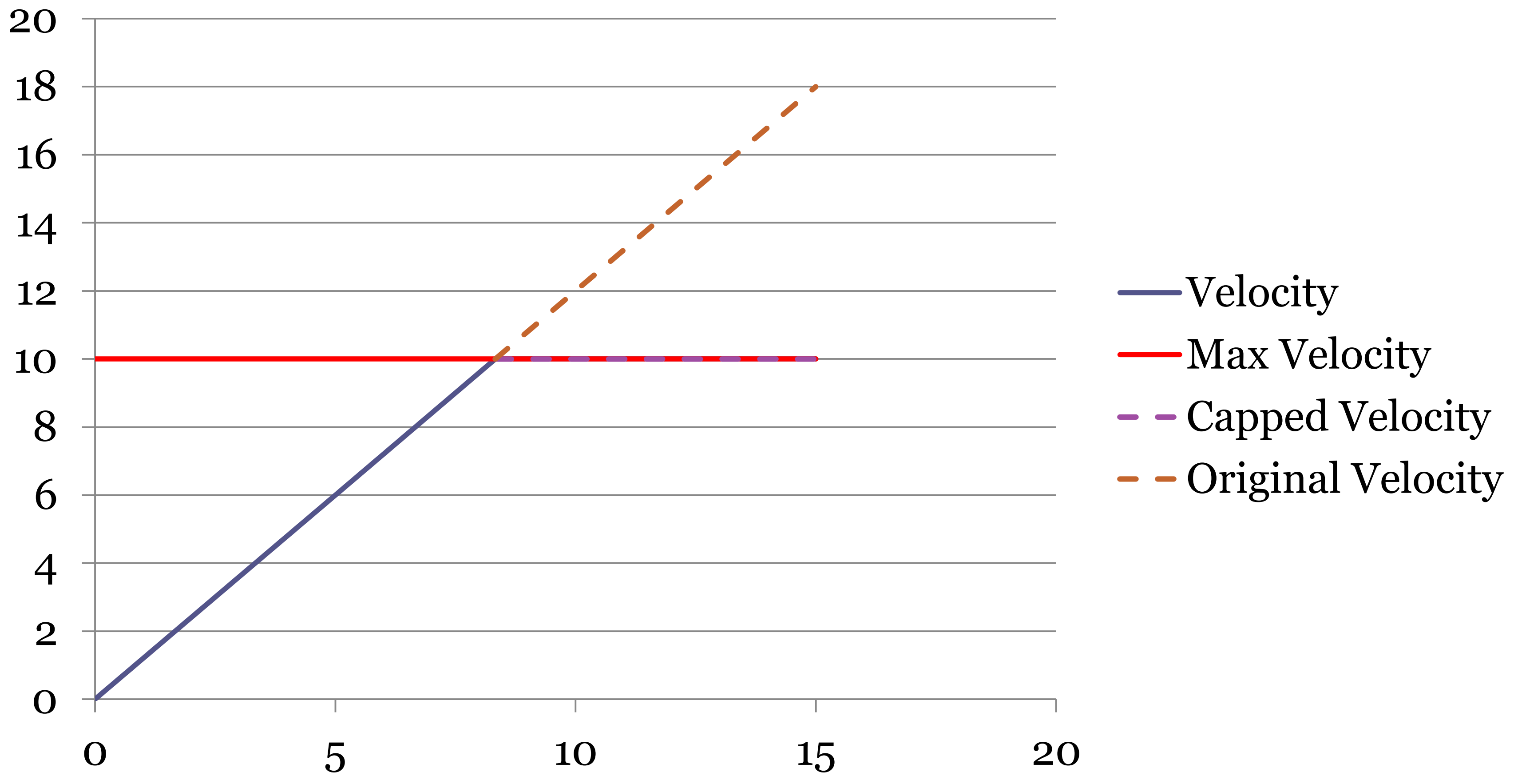
$$newVel = a * dt + currVel$$
$$newPos = newVel * dt + currPos$$

- The second one has less computations.
- The second one is true, because within dt we consider our moves are linear.
- Both allow us to manipulate the velocity before updating the position:
    - Set a velocity cap
    - Simulate friction
    - Etc..

Asteroids – Ship's Velocity Cap (2/4)

- A velocity cap can be set in different ways
- Simplest:
  - Set a maximum velocity magnitude
  - Every time a new velocity is computed, compare its magnitude to the maximum
  - Greater? Set it to the maximum
- Works, but feels unrealistic
  - Reaching the maximum velocity is instantaneous
  - Maximum velocity is not reached smoothly

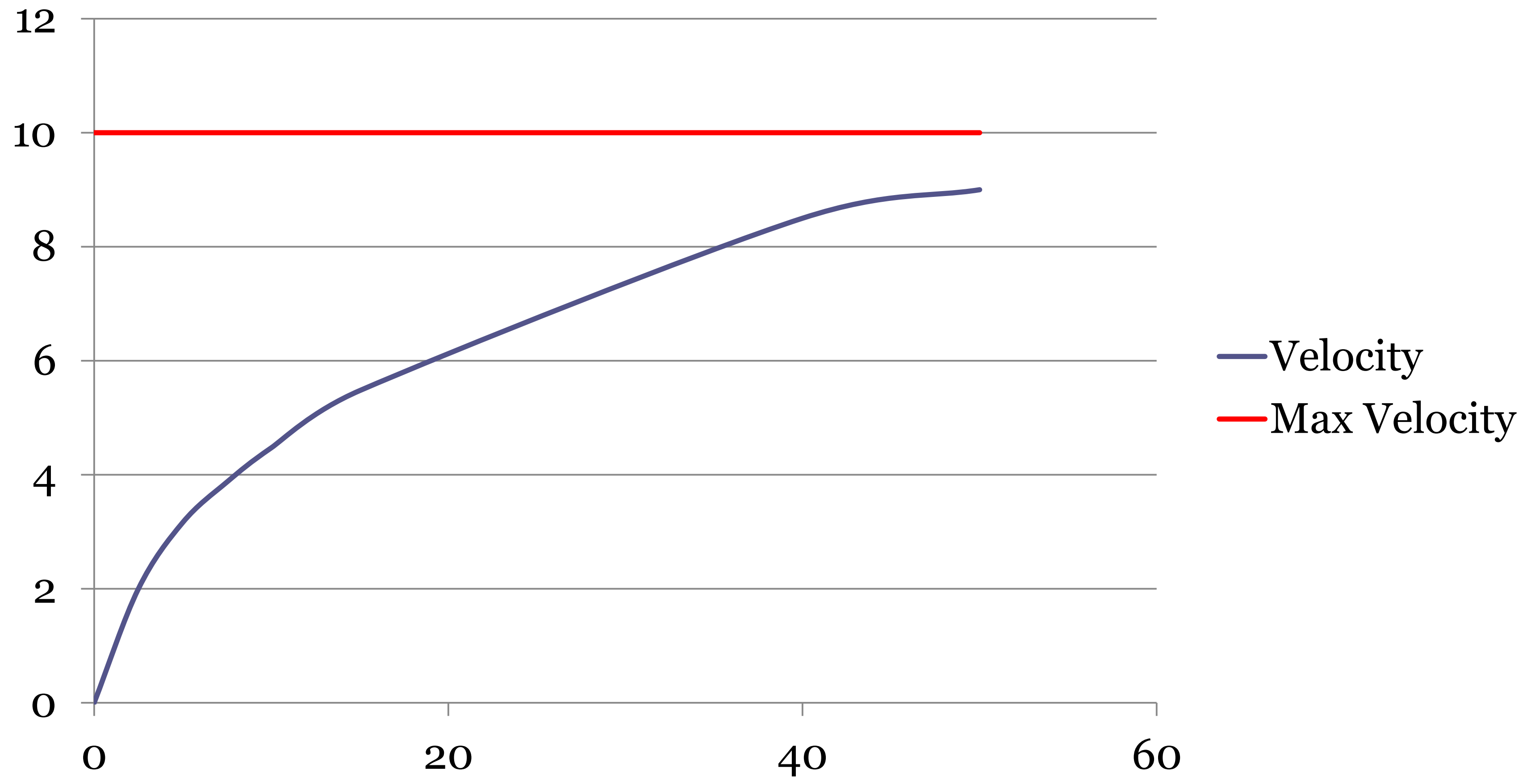
Asteroids - Ship’s Velocity Cap (3/4)



Asteroids - Ship’s Velocity Cap (4/4)

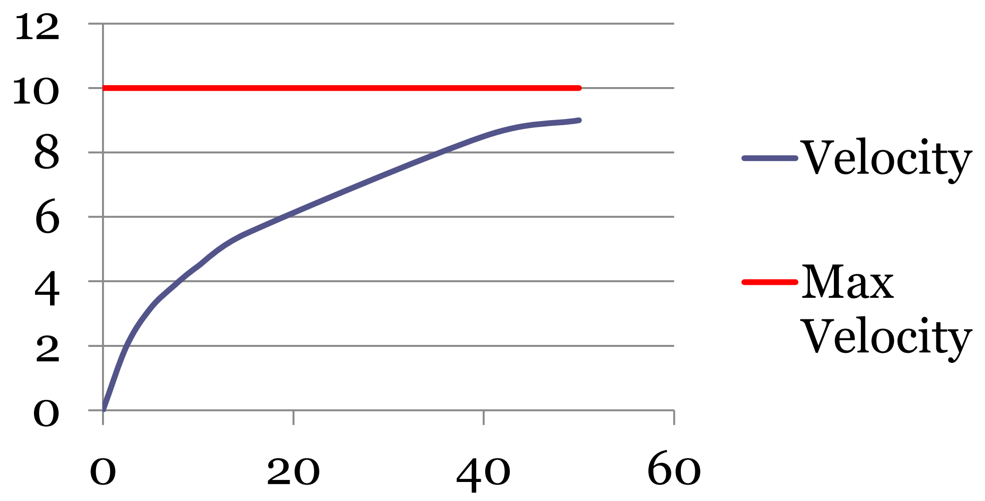
- In reality, maximum velocity is reached due to **friction**
- Friction is a force
  - Accelerations are derived from forces
  - Velocities are derived from accelerations
  - Conclusion: Velocities are affected by friction!
- Friction allows objects to smoothly reach their maximum velocities

Asteroids – Better Velocity Cap (1/6)



Asteroids - Better Velocity Cap (2/6)

- In CS230, we're not using forces
  - Accelerations and velocities are directly assigned
  - This means that we can't apply friction
  - But we still want to achieve the following result:



Asteroids – Better Velocity Cap (3/6)

- Friction will be emulated
- There are different techniques to achieve a smooth velocity capping
- Previous Implementation

$$newVel = a * dt + currVel$$

$$newPos = newVel * dt + currPos$$

Asteroids – Better Velocity Cap (4/6)

- Our velocity capping technique

$$newVel = a * dt + currVel$$

- New step: $newVel = newVel * 0.99$

$$newPos = newVel * dt + currPos$$

- Isn't that just reducing the velocity by 1%?

Asteroids – Better Velocity Cap (5/6)

dt = 1	Frame 1			Frame 2			Frame 3	
	Original	*0.99		Original	*0.99		Original	*0.99
Given: currPos	(0;0)	(0;0)		(2;3)	(1.98;2.97)		(6;9)	(5.92;8.88)
Given: currVel	(0;0)	(0;0)		(2;3)	(1.98;2.97)		(4;6)	(3.94;5.91)
Given: a	(2;3)	(2;3)		(2;3)	(2;3)		(2;3)	(2;3)
Computed: newVel	(2;3)	(1.98;2.97)		(4;6)	(3.94;5.91)		(6;9)	(5.88;8.82)
Computed: newPos	(2;3)	(1.98;2.97) 99% of the original value		(6;9)	(5.92;8.88) 98.6% of the original value		(12;18)	(11.8; 17.7) 98.3% of the original value

Asteroids – Better Velocity Cap (6/6)

- Every frame, the velocity is reduced by a greater %
  - Feels realistic
  - Maximum velocity is reached smoothly

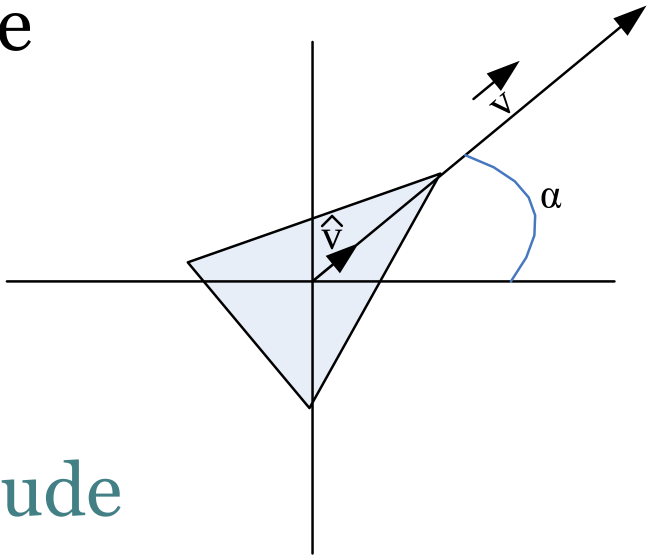
Creating Bullets (1/2)

- For simplicity, bullets will be created at the same location of the ship
  - Which means the ship's current position is needed
- Bullets are not accelerated
  - They have a constant velocity
  - That velocity has a predefined magnitude
    - Similar to the ship's predefined acceleration magnitude
  - Problem: Computing the direction of the bullet's velocity

Creating Bullets (2/2)

- Computing a newly created bullet's direction is similar to computing the ship's acceleration
  - They both depend on the ship's orientation α
  - Compute $\hat{v}$, which is equal to $\hat{a}$
  - Scale $\hat{v}$ by the predefined magnitude in order to get $\vec{v}$

$$\vec{v} = (\hat{v}.x * 200 + \hat{v}.y * 200)$$



Creating Asteroids

- Asteroids, like bullets, have constant velocities
- The 2 differences:
  - Asteroids are created at random locations (preferably outside the viewport, or at a destroyed asteroid's last position), while bullets are created at the ship's current position
  - Asteroids' velocities have "random" directions, while bullet velocities' direction depend on the ship's orientation