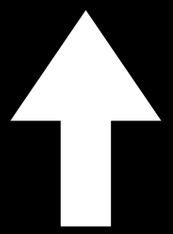
PC (and Console) Game Development

01 - Introduction

Real-time Agent-Based

Interactive Simulations



Interactive Simulations

- Mathematical equations.
- Most of them are complex differential equations (or difference equations) that are solved through integration.
- Most of the mathematical details are abstracted when programming (thankfully).

$$d\vec{p}(t) = \big(1 - I(\vec{p}(t))\big)d\vec{p}(t) + I(\vec{p}(t))\vec{r}_{\vec{n}}\big(d\vec{p}(t)\big)$$

Looks painful, but fortunately we won't be dealing with this most of the time.

Code-translation of Mathematical Models

$$d\vec{p}(t) = \big(1 - I(\vec{p}(t))\big)\, d\vec{p}(t) + I(\vec{p}(t))\vec{r}_{\vec{n}}\big(d\vec{p}(t)\big)$$

is loosely translated to

```
Vector2 getVelocity(double time) {
  CollisionInfo c = getCollisions(position);
  if ( c != null ) {
   velocity = Vector2::reflect(velocity, c.normal);
  }
  return velocity;
}
```

Integration

$$\vec{p}(t_0) = \int_0^{t_0} \big(1 - I(\vec{p}(t))\big) d\vec{p}(t) + I(\vec{p}(t)) \vec{r}_{\vec{n}}\big(d\vec{p}(t)\big) dt$$

Can be solved through something called “Euler Integration” (a kind of numerical integration algorithm) instead of doing it Ma20.2/Ma21 style. (yay?)

```
Vector2 getPosition(double time, Vector2 initialPosition, Vector2
initialVelocity) {
  position = initialPosition;
  velocity = initialVelocity;

  for ( double i = 0; i < time; i += 0.001 ) {
   position += getVelocity(time) * 0.001;
  }
  return position;
}
```

Simulating the Game Model

- The Euler Integration algorithm won't give an exact solution to the game/mathematical model. It is an *approximation.*
- Approximation is one of the powerful tools you have as a game developer.
  - It doesn't matter if it's not realistic or inaccurate *as long as it looks ok.*
  - e.g. You can't change direction mid-air but it seems more natural to do so.

Approximation Caveats

- Usually, you don't need to know the error bounds of your approximation.
- However, sometimes, especially in physically-based simulations, approximation errors build up and observable errors (i.e. "bugs" from the player's perspective)
- A class of bugs you have to watch out for when doing numerical simulations is *numerical stability.*

Game Categories

Real-Time	Turn-Based	
First-Person	Third-Person	Abstract
Single-Player	Multi-Player	Massively-Multiplayer
2D Gameplay	3D Gameplay	
2D Graphics	3D Graphics	
Physically-Based		

Game Components

AI

Entity Framework

HUD/GUI

Animation

Collision and Physics

Graphics

Audio

Input

Data Structures and Algorithms

Configuration Manager

Platform Independence

Memory Manager

Resource Manager

Game Development Team Structure

- Designers
- Programmers
- Artists
- Producers
- Quality Assurance (testers)

Game Programming Specialization

- Gameplay
- Graphics
- AI
- Physics
- Tools
- Audio
- Architecture (usually the Senior Programmer)