Weather Effects (Group 1)

Jared Headings, Ted Zhu, Ian Kirchner

Weather in Games

Audio and Visual Effect

Direct Change to Play

Weather as a Visual Effect

Weather Affecting Play - Both Simple, and Complex

https://youtu.be/PeT8hU6Jo0I?t=22s

GTA V:

- Highly complex weather system!
  - Default: Cloudy weather, no adverse effects on gameplay
  - Every 2-4 in-game days: **Sunny** weather, no adverse effects on gameplay
  - Every 3-5 in-game days: **Rainy** weather of random intensity

- Rainstorms in GTA are masterpieces
  - Water pools up on low spots in roads
  - Less stability in vehicles/risk of hydroplaning
  - Will start as a drizzle, gradually build in intensity, then taper off
  - Lights will dynamically turn off in whole sections of the game, simulating power outages
  - Controller vibrations during thunder

Complexity can come at a price...

- Can be expensive
- Time consuming to implement
- Can cause massive FPS drops...

So *why* do we have weather in games?

Weather == Immersion

- “The more familiar everything feels, the less we notice the machinery behind the illusion.”
- The mystical “suspension of disbelief”
- DriveClub - https://www.youtube.com/watch?v=hViwrRGfuHU

Immersion

- Games lacking immersion can make it harder for you to suspend your disbelief

- Physics-based, realistic weather is a great way to gain immersion!
    - It's what we experience every day.

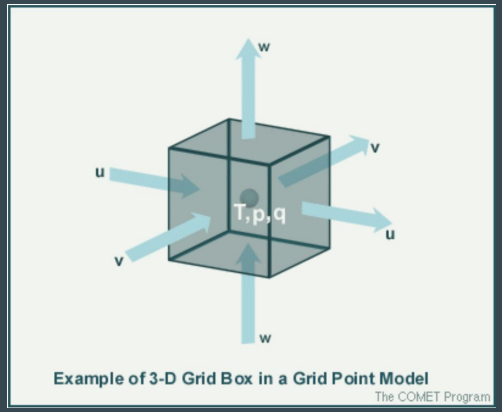
Creating Realistic Weather

100% realism = 100% immersion

Modeling Weather

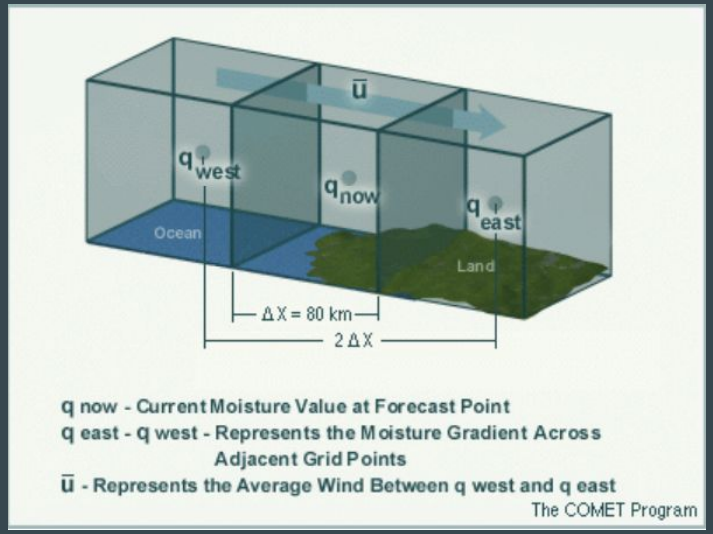
Weather is a highly complex system

Requires flow modeling along a 3D grid



Example of 3-D Grid Box in a Grid Point Model

The COMET Program



q now - Current Moisture Value at Forecast Point

q east - q west - Represents the Moisture Gradient Across Adjacent Grid Points

ū - Represents the Average Wind Between q west and q east

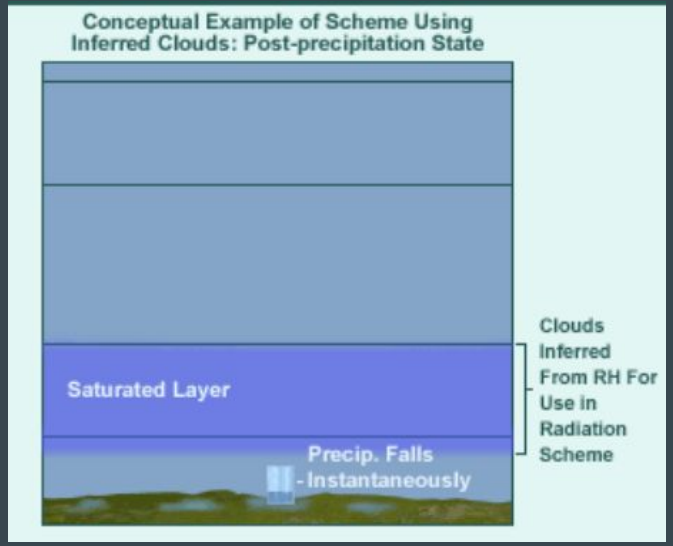
The COMET Program

Modeling Weather

- Large number of factors to model
  - Relative Humidity
  - Temperature
  - Wind
  - Convection
  - Radiation
- Many different scales to consider
  - Global
  - Regional
  - Local

Parameterization

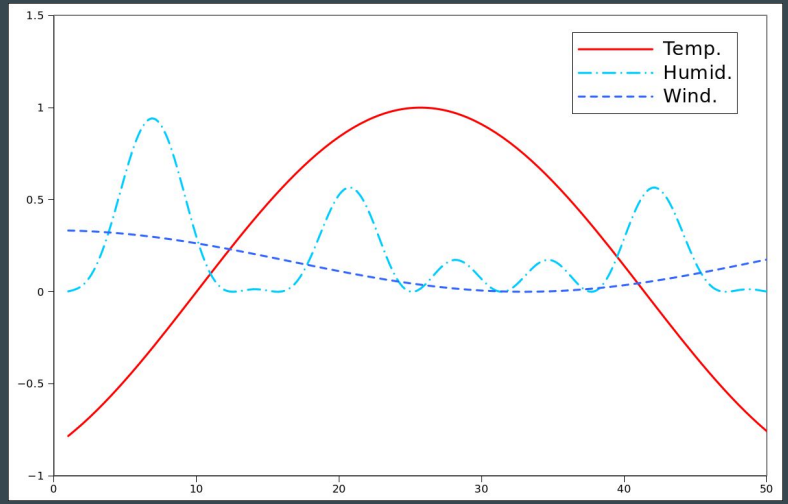
- Resolution can only get so high
  - Computationally impossible at certain point
- Large scale simulation requires supercomputers
- Highest resolution uses grid cells of ~5km^3
- Requires Parameterization
  - Abstracting the internal processes of cells

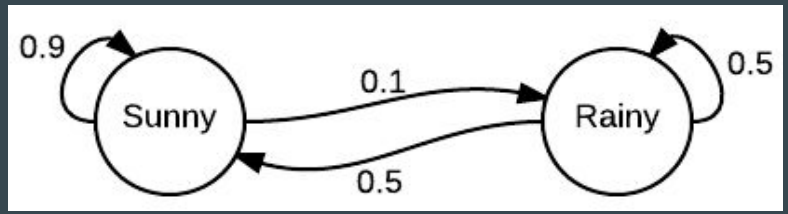


IAN DEMO

Modeling Weather in Games

- Add Levels of Abstraction
  - Static weather
  - Markov Model
  - Temperature/Humidity/Wind curves
- Easier to create, easier to simulate
- Provides a “good enough” approximation



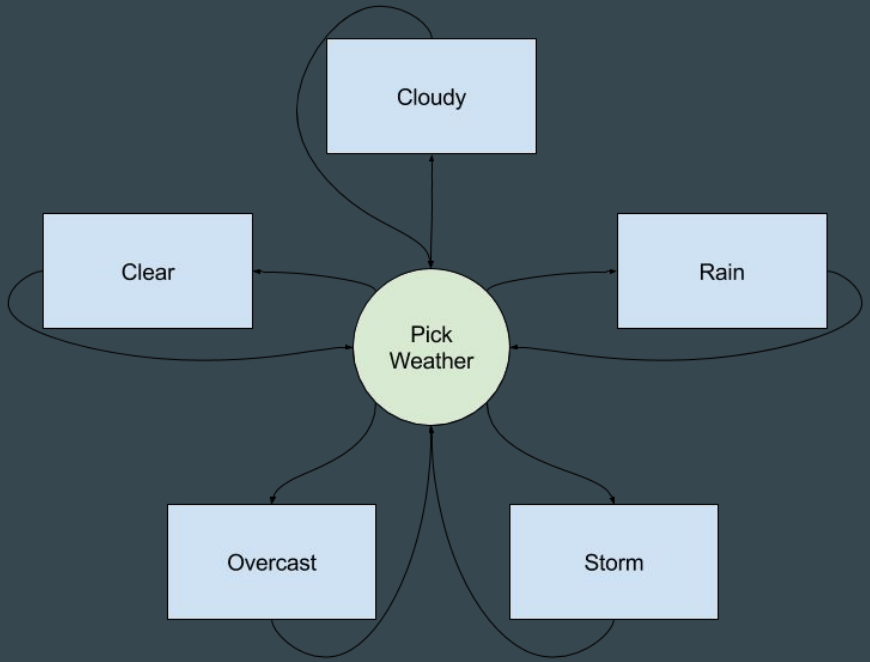


Static Weather

- Predefine weather for areas in the game
- Pros
  - Allows for the most control
  - Easiest to implement
- Cons
  - Not adaptable
  - Detracts from a sense of a living world

Markov Model

- Define all weather states
  - Clear
  - Cloudy
  - Overcast
  - Rain
  - Storm
- Define each state's transition probability
- Transition to new weather on a timer

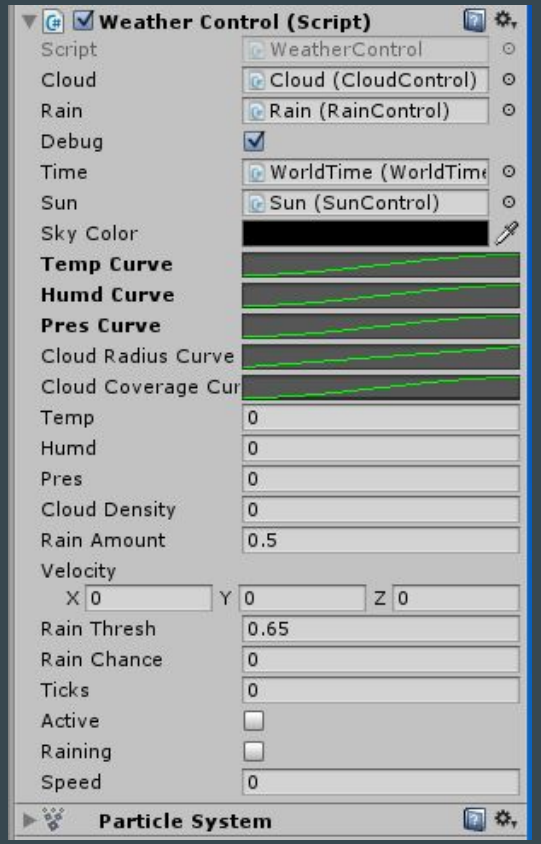


Markov Model

- Pros
  - Provides control while still having weather transitions
  - Not very complex
- Cons
  - Difficult to find average rain chances
  - Requires additional interpolation for weather transitions

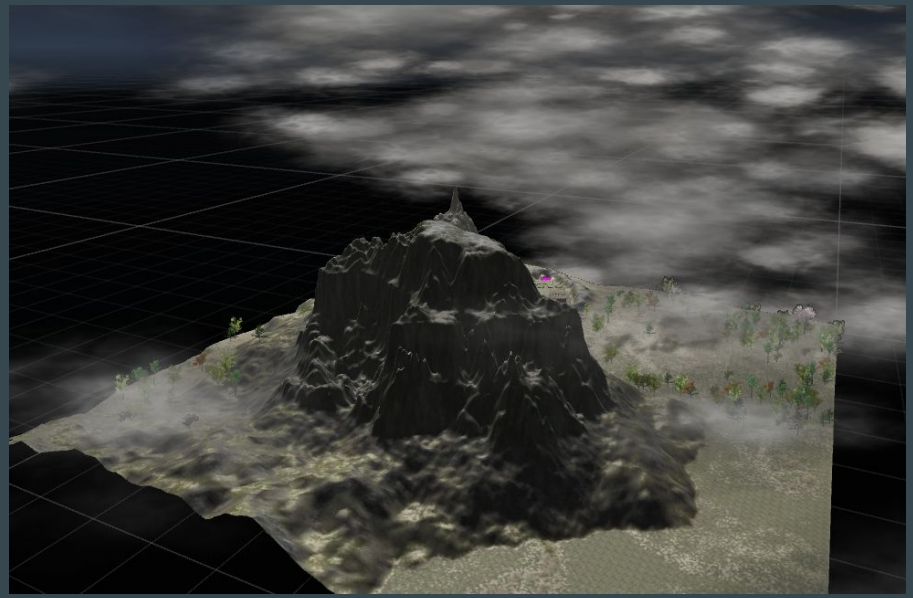
Weather Feature Curves

- Create or generate curves for various features
  - Wind
  - Temperature
  - Humidity
  - Etc.
- Sample these at a given rate
- Combine the samples for weather
  - Temp > 0 ^ Humd > 65 => rain



Weather Feature Curves

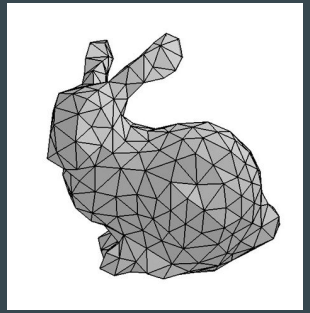
- Pros
  - Transitions are more smooth
  - Capable of more realism
- Cons
  - Difficult to control
  - Difficult to implement



Physically Based Simulation

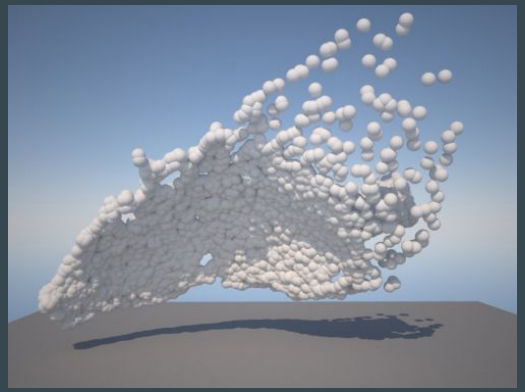
Representing Physical Objects

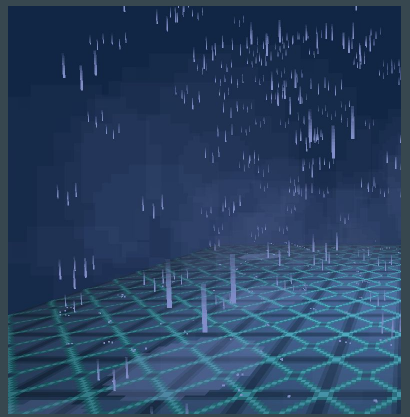
- Meshes
  - used in 3D games to represent solid objects.
  - computationally expensive in simulations
- Sprites
  - used in 2D games
  - manually programmed without physics-based simulation
- Particle Systems
  - simple images or meshes moved in great numbers
  - depict entities that are fluid and intangible in nature
  - e.g., liquids, smoke, flames, clouds



Particle Systems, in depth.

- Each particle represents a small portion of a fluid
- All the particles together form an impression of the complete entity being represented
    - E.g., A cloud, tornado, mist

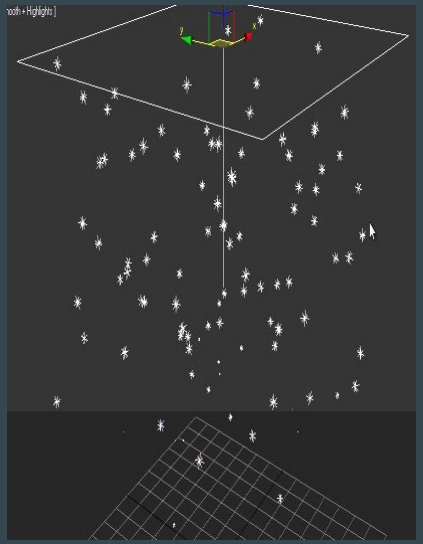




Particle Systems vs. the Particle

Particle System

- A Particle Emitter
  - *Some 3D mesh or shape*
- Emission rate
  - *particles/second*
- Emission position on mesh
  - *Usually randomized*

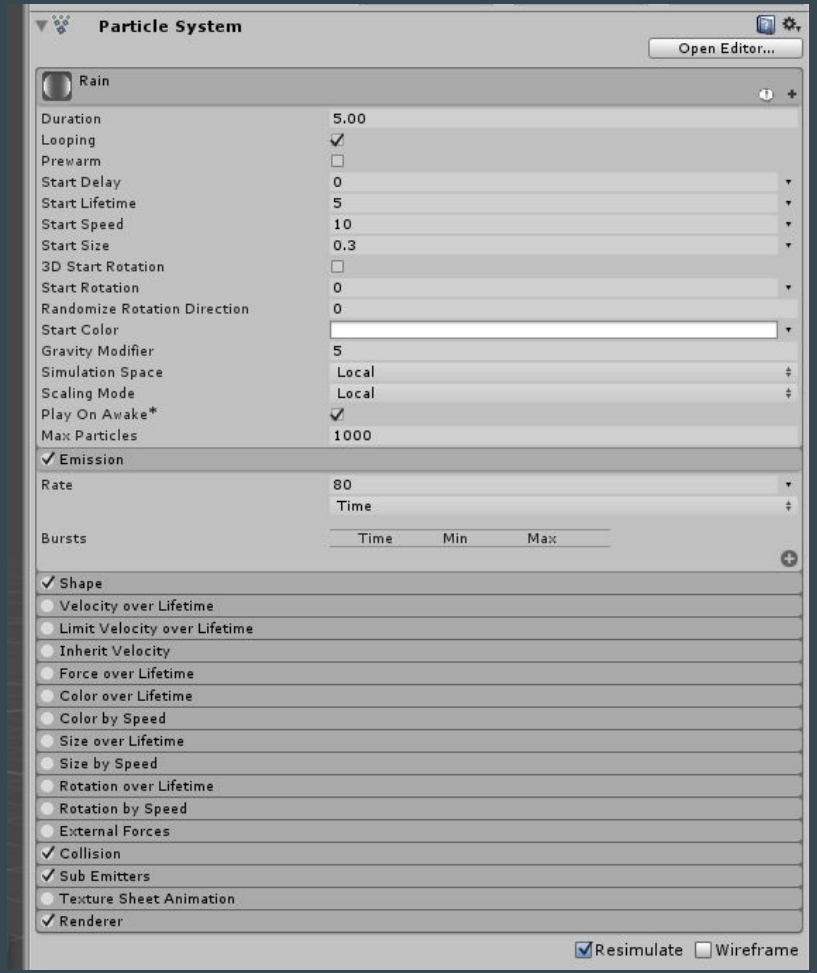


Particle

- Lifetime
  - *Usually a few seconds*
- Velocity vector
  - *Affected by forces acting on the system*
- Appearance
  - *Size, color, rotation*

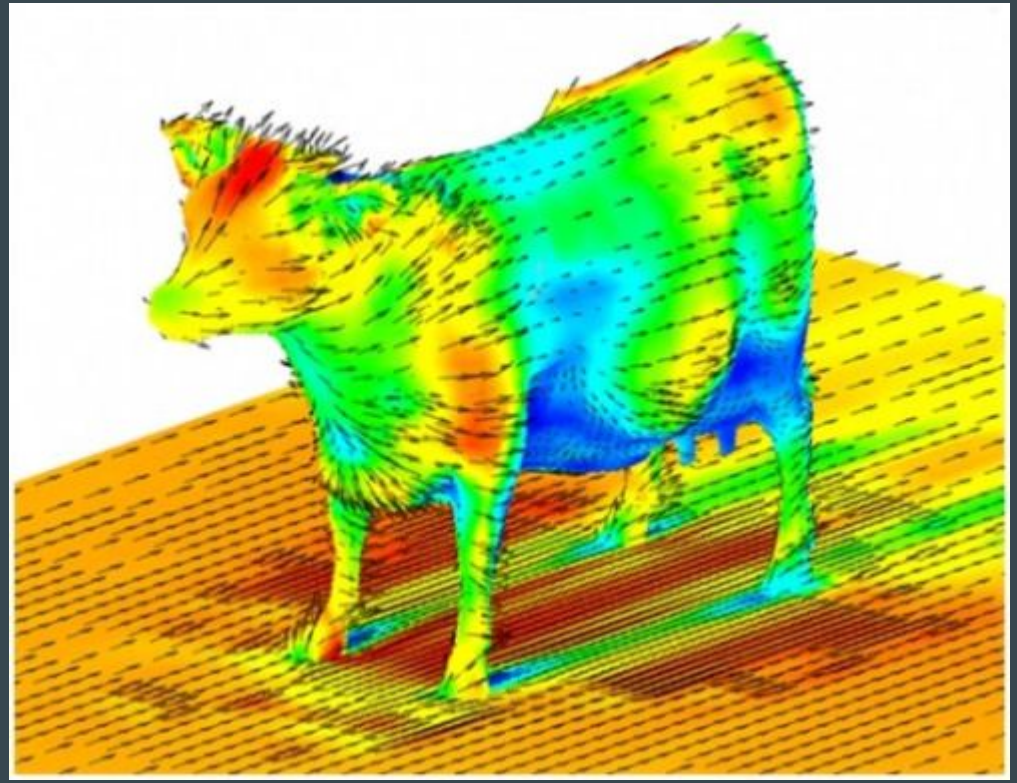
Particle Systems in Unity

- A Particle System can act as its own GameObject,
- Or a Component of an existing GameObject.
- Unity provides several ways to customize its behavior.
- Caveat: Cannot apply user-defined forces on given particles.



Custom Particle Systems

- Need to create own Particle Object to perform arbitrary force manipulations on particles.
- Custom particle system needed for simulation of a tornado.



Fluid Dynamics Simulation

"DYNAMICAL MODELING OF A TORNADO"
Steven Torrisi, 2015

$$\vec{u}_t + (\vec{u}\cdot\nabla)\vec{u} = \frac{1}{Re}\Delta\vec{u} - \nabla p + f$$
$$\nabla\cdot\vec{u} = 0,$$

where u is the fluid velocity vector,

p is the pressure,

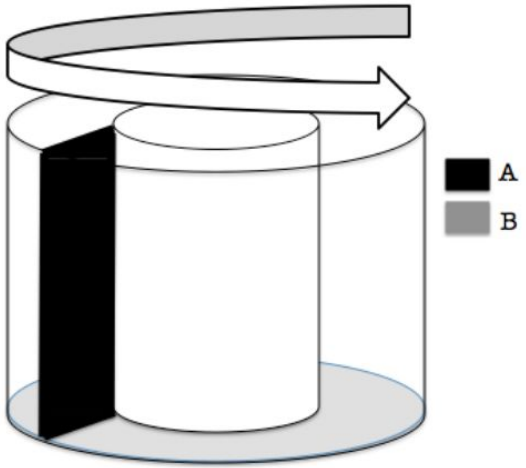
f are other body forces such as gravity,

and Re is the Reynolds

number (viscosity)

$$\int_S \mathbf{n}p dS = -\rho\int_S (\mathbf{n}\cdot\mathbf{q})\mathbf{q}dS \qquad (2.2)$$

Which states that the pressure applied onto an arbitrary surface S (by multiplying the vector pointing inwards, $\mathbf{n}$ by the pressure p) is equal to the momentum flowing into or out of the system through the same surface.



Custom Particle Systems

```
void FixedUpdate () {

    lifetime += Time.fixedDeltaTime;
    if (lifetime > 20)
    {
        Destroy(this.gameObject);
    }

    Vector3 towardsCenter = Vector3.Normalize(vortextCenter - transform.position);

    Vector3 towardsCenterForce = towardsCenter * 10;

    Vector3 tangential = Vector3.Normalize(Vector3.Cross(towardsCenter, Vector3.up));

    Vector3 tangentialForce = tangential * 4f;

    Vector3 upwardsForce = Vector3.up * Random.Range(18, 25);


    Vector3 force = towardsCenterForce + tangentialForce + upwardsForce;

    rb.AddForce(force);

}
```

TED DEMO

Bibliography

http://gamestudies.org/0801/articles/barton

http://docs.unity3d.com/Manual/ParticleSystems.html

http://rams.atmos.colostate.edu/at540/fall03/fall03Pt7.pdf

https://www.assetstore.unity3d.com/en/#!/content/2714

https://gamedev.stackexchange.com/questions/20564/algorithm-for-randomized-weather