🤖 Control Rig

Control Rig

Forward Solve

Helps in Making animations from a rig, whereas the Backward solve helps editing existing animation

1. Controls and Bones

Create a control for a bone, drag it to forward solve as GET CONTROL. Drag it's parent bone you want to manipulate and click SET BONE, pin in transform from control to bone node and connect it to the FORWARD SOLVE.

NOTE - Controls are generally not child of any bones, because moving the bones will make them move with them as well…

2. Inverse Kinematics

A way to manipulate bones from the child to parent, whereas in forward kinematics that is default bones are manipulated from parent to the child bone

- Basic IK node
    - Item A and B - effector child of b child of a, 3 bones we want to control
    - Effector Item - third bone the bottom most
    - Effector Vector - The control generally that controls end of hand for example
    - Primary axis - unit vector of Direction of item b bone
    - Secondary axis - unit vector of direction of item b to pole vector
    - Pole Vector - Item B's (ex knee) control's location

3. Create an animation

Drag control rig in scene, key frame in the sequencer as it opens automatically. Right click on the left bar at control rig track -

- Bake animation sequence

Creates a baked animation sequence from the sequencer

- Create linked animation sequence

Creates a linked animation sequence that can be edited as we edit the animation further in the sequencer.

4. Aim at

Aim node helps to aim the specific location or direction, we can rotate the spine of the character to aim at a specific location/direction. Put weights for 3 different spine bones like 0.25 0.50 and 1.00 to smooth out the rotation.

- AIM node
    - PENDING….

5. Aim at through Blueprints

Instead of the AIM NODE spine target being a control, we set it to a variable. Drag control rig in animation bp, set the value of this variable (expose the variable from control rig first). For the third person character the value to look at can be the camera forward vector direction.

6. Limiting controls

Controls can be limited, for example a control should not move out of a box, or head shouldn't move out of -70 to +70 degrees.

7. FABRIK node

Takes an array of bones as input, and an affector transform to lean the bones towards the affector target.

8. CCDIK node

The same as FABRIK just gives a rotation limit for each or all bones at once, so character doesn't bend too much if the door knob is far away.

9. Walking Spider (Procedural Animation)

- Full Body IK (FBIK)

- Unlike Basic IK, FBIK treats the whole skeleton as a web. Moving a leg can "pull" the body.
- Root: Usually the Pelvis.
- Effectors: The tip of each leg.
- Chain Depth: Set to 3 (for a 3-joint leg) to ensure the leg bones move before the body starts to tilt.

- The "Locked" vs "Target" System

- worldLockedTransforms: The "Starting Line." Where the foot was at the beginning of the step.
- worldTargetTransforms: The "Finish Line." The coordinate in the world where the foot wants to land.
- The Swap: Inside the Trigger function, we set Locked = Target (Old) before calculating the New Target. This ensures the leg starts its next step from exactly where it just landed.

- Alternating Tetrapod Gait (Syncing)

- Prevents the spider from "hopping" with all legs.
- Groups: Split 8 legs into two groups: Group A [0, 3, 4, 7] and Group B [1, 2, 5, 6].
- Logic: A timer flips a Boolean every 0.2s–0.25s. Leg Trigger = (Distance > Threshold) OR (Is My Group's Turn).

- Velocity Projection (The "Treadmill" Fix)

- If a spider moves fast, a static target (e.g., 200 units ahead) isn't enough. The body will outrun the foot.
- The Fix: New Target = Current Position + (Velocity * Step Duration). This "over-predicts" the landing spot so the foot stays under the body longer.

- Surface Normal Alignment

- Allows the spider to tilt its body on slopes/walls.
- Raycast (Line Trace): Shoots a ray from the pelvis to the floor to get The Hit Normal.
- From Two Vectors: Compare "Vector Up" (0,0,1) to the Hit Normal. It creates a rotation offset that is multiplied into the Pelvis rotation.

- TROUBLESHOOTING & CRITICAL FIXES

- The "Double-Jump" Jitter: Never measure the distance from the Locked (Start) position. As the spider moves, the start point gets further away, causing the leg to trigger repeatedly. Always measure distance from the Target (Landing) position. Once the foot lands, the distance becomes 0, and the logic stabilizes.

- Spring Interp Logic:
  - Stiffness: How fast it tries to catch the target.
  - Damping: 1.0 = smooth stop. < 1.0 = wobbly/overshoot.
  - Used on the Pelvis to make the body "lag" behind turns, creating a sense of weight and organic "juice."

Environment

Lighting

Environment Light Mixer

Environment Light

1. Directional Light (The "Sun / Moon")

- **What it is:** A light source infinitely far away that shines parallel light rays across your entire level.
- **What it does:** Creates direct sunlight, harsh shadows, and defines the time of day.
- **Interdependence:** By itself in a blank level, it only lights up faces directly pointing at it—everything in shadow stays pitch black.
- **Atmospheric SunLight Index:** It is used to create 2 directional lights in the game, essentially 2 suns or a sun and a moon at the same time.

2. Sky Atmosphere (The "Air / Gas")

- **What it is:** A physically based simulation of Earth's atmosphere (air density, Rayleigh scattering, Mie fog).
- **What it does:** It turns the void into a blue sky or warm sunset.
- **Interdependence: It does nothing on its own.** It *needs* a Directional Light to act as the sun. Once there is Directional Light in the Scene, the air catches the sun's rays and lights up the sky.
- **Requires:** directional light to work.
- **Atmospheric Fog:** the older version of it, sky atmosphere does not work with the Atmospheric Fog.

3. Volumetric Cloud (The "Clouds")

- **What it is:** 3D rendered, volumetric cloud structures generated in real-time.
- **What it does:** Renders clouds in the sky that cast real shadows on your ground.
- **Interdependence:** Receives light from the **Directional Light** (Sun) and alters what the **Sky Light** captures for ambient shadows.

4. Exponential Height Fog (The "Distance Haze")

- **What it is:** A layer of haze or mist at the bottom of your map.
- **What it does:** Blends the ground level into the horizon so distant objects look naturally faded.
- **Interdependence:** When connected to the **Sky Atmosphere**, it takes on the exact sun and sky color automatically.

5. Sky Light (The "Ambient / Bounce Fill")

- **What it is:** A dome-shaped light that captures everything in the sky and projects soft, ambient light down into the dark shadows.
- **What it does:** Prevents shadows from being 100% pitch-black.
- **Interdependence:** It relies on the **Sky Atmosphere** or **Volumetric Clouds** existing first. If you have **Real-Time Capture** checked on the Sky Light, it reads the color of the Sky Atmosphere and bounces blue/grey ambient light back onto your level.

6. Atmospheric Fog (The "Legacy Air Haze")

- **What it is:** The original, legacy fog and sky haze actor used in older versions of Unreal Engine (prior to UE4.24).
- **What it does:** It simulates simple Rayleigh scattering in the air to give the horizon a hazy, distant look and tints the background based on a directional light.
- **Interdependence:** It relies on a **Directional Light** to calculate where the horizon horizon light is coming from.
- **Conflict Notice: Do not use this alongside `Sky Atmosphere`.** It is an obsolete system that conflicts with modern atmosphere rendering, causing flickering or weird overlapping fog calculations.

7. `BP_Sky_Sphere` (The "Paint-on-a-Dome Sky")

- **What it is:** A pre-built blueprint containing a static, inverted dome mesh with a custom sky material wrapped around the inside.
- **What it does:** Renders a simple, non-physical sky texture with customizable cloud colors, sun position, and horizon gradients.
- **Interdependence:** It relies on a **Directional Light** being manually selected in its Details panel under `Directional Light Actor`. You then have to click `Refresh Material` so it snaps the visual sun texture to match your light's rotation.
- **Conflict Notice:** It is a legacy shortcut. If you are building a realistic modern environment using **`Sky Atmosphere`**, you should **delete or hide `BP_Sky_Sphere`** to avoid having two sky meshes fighting each other in your scene.

8. Exponential Height Fog (The "Ground & Horizon Mist")

- **What it is:** A volumetric fog actor that places a dense layer of haze or mist at the bottom of your level that gradually thins out as you go higher in altitude (hence "exponential").

- **What it does:** It gives your level depth by making distant objects look faded, creates thick valley/ground fog, and generates atmospheric light shafts ("god rays") when light passes through it.
- **Interdependence:** It works completely on its own, but when a **`Sky Atmosphere`** is present, it automatically samples the sky and sun colors to blend ground fog naturally into the horizon.
- **Conflict Notice:** If used alongside legacy **`Atmospheric Fog`**, the two fog density calculations will overlap and create thick, double-layered visual artifacts. It should be paired with `Sky Atmosphere` instead.

Reflections

Reflection Captures

- ## Box Reflection Capture

  Captures reflections around the environment as much as the size is given, projects those reflections on reflective surfaces.

  - ➔ Lumen does it automatically, not needed when using Lumen
  - ➔ UE4.27 uses Screen Space Reflections (those can be turned off from the post process volume settings), it’s a capture of the whole world and projected on reflective surfaces. For precise control turn it off and build reflection captures.
  - ➔ Moveable meshes are needed to tell specifically VISIBLE IN REFLECTION CAPTURE, still it might not be visible. Best for Static Meshes.

- ## Sphere Reflection Capture

  Same as box reflection capture, just captures a spheric shape, better if reflective meshes are round.

- ## Planer Reflection Capture

  Almost the same as the upper two, the difference is it’s heavy on GPU and captures in real time and puts reflections on reflective surfaces.

🎨 Materials

Materials

- Material
- Material Instances
    - Changing parameters via Blueprints
        1. Material parameter collection
        2. Dynamic material instance
        3. For mesh component, node “Set scalar parameter value on Materials node”

- Material Functions:

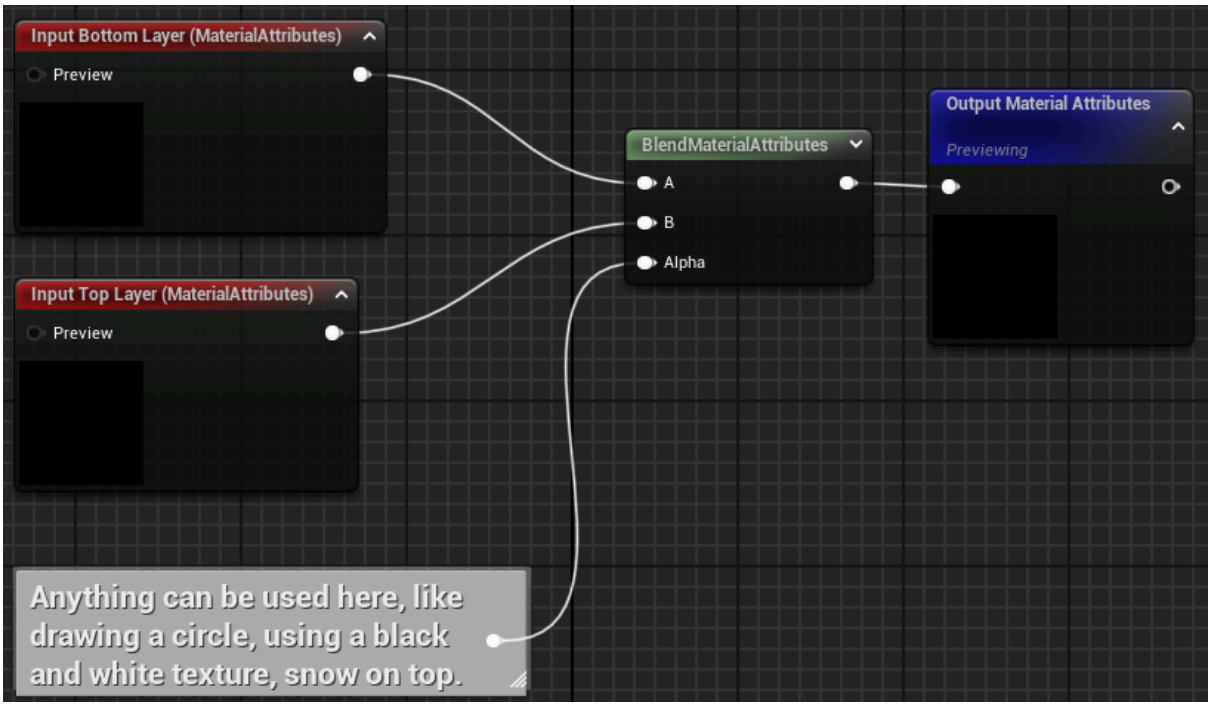
  A complex logic can be put into a material function and reused again and again as a single node. **Material Function Instances** can alter variables in its parent material function to create a different but same logical material function.

- Material Layers:

  These are like materials itself, but as layers those can be blended based upon the Material Layer Blend. **Material Layer Instance** is self explanatory.
  Material layers use material attributes, so plug in make material attribute node to get roughness, albedo and others.

- Materials Layer Blend:

  Material layer blends are used to Lerp between Top and bottom layer, all the logic of blending relies here as shown in the picture.
  Material Layer Blend Instance is self explanatory.



- ## Material Parameter Collection:

  Material parameter collection holds variables that can be changed from any blueprint and changes its value throughout all the materials it's used in. It is used like a global variable for example wind speed.

- ## Subsurface Profile:

  How lights bounce inside a mesh. For example blood can be seen from our skin. It's just a combination of variables that can be used in a material after setting its shading model to Subsurface profile.

- ## Neural Profile:

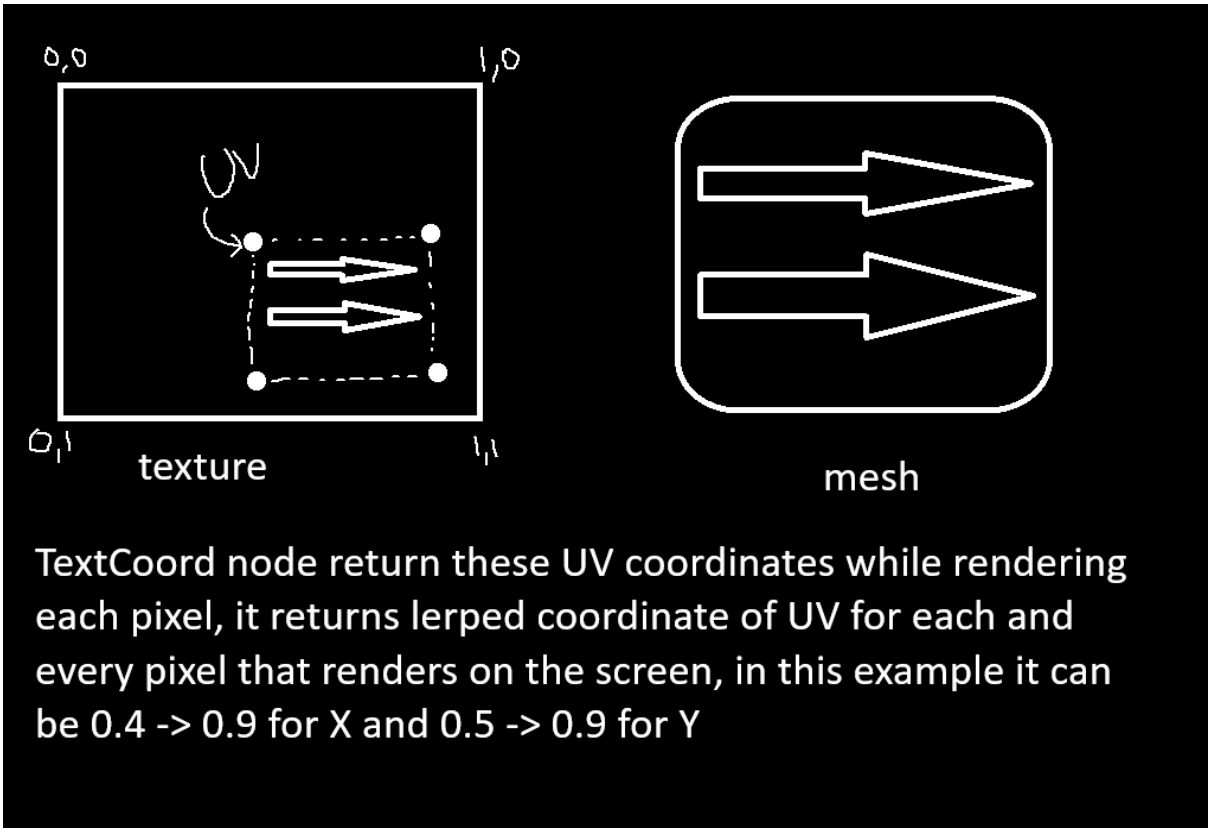
  An ONNX Neural Network model can be imported to process screen output at the very end and create effects like hand drawn, they are more precise than basic toon post process effects.

- ## Specular Profile:

  Requires Substrate Material Framework to be turned on. Specular is the reflection of the light source itself on a surface. It can be used to control the way this glint shows up from different angle for example how a CD glows irl. It uses a texture to mimic that CD glow.

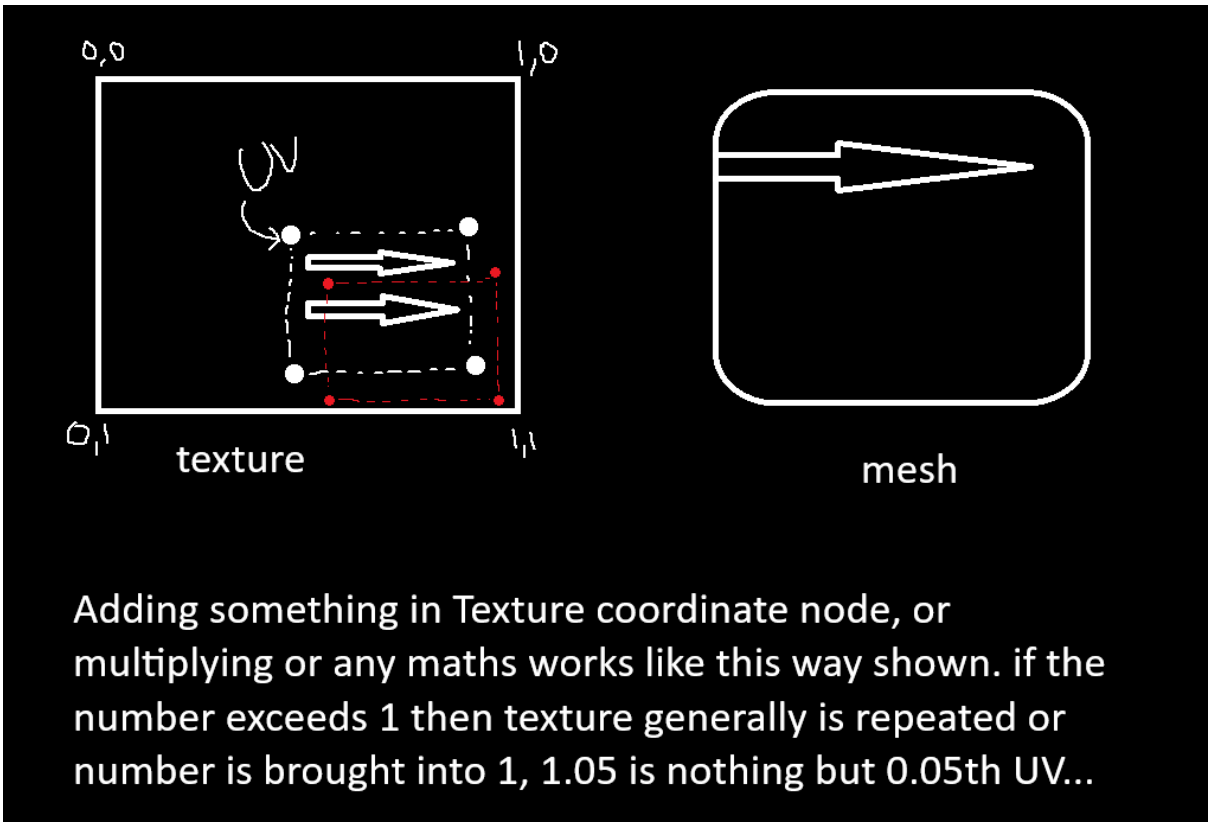
Altering UV's

- Scaling Textures:
    - Texture Coordinate Node:



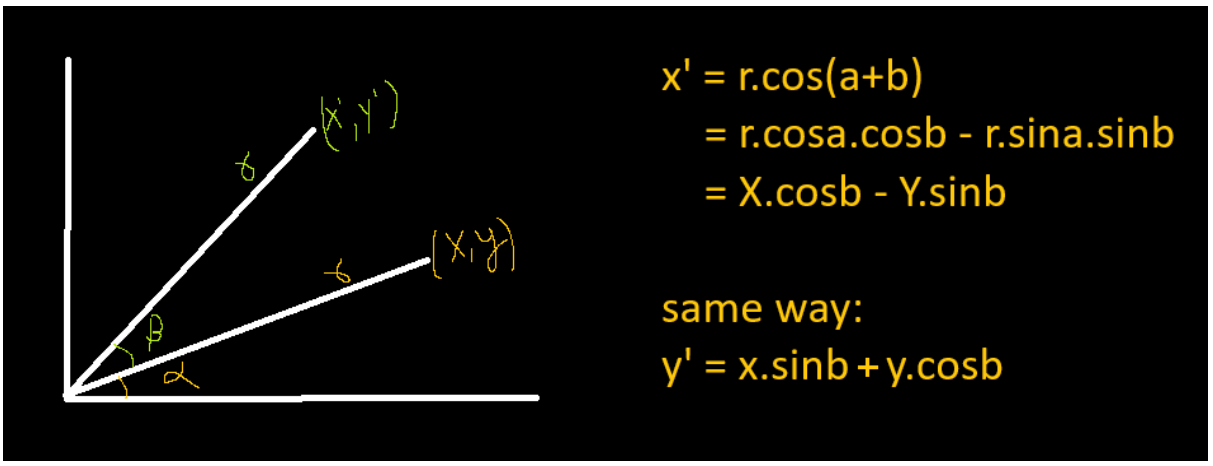
1) It returns a vector2D
    a) RG
    b) R = X coordinate of texture
    c) G = Y coordinate of texture

- Move UV:



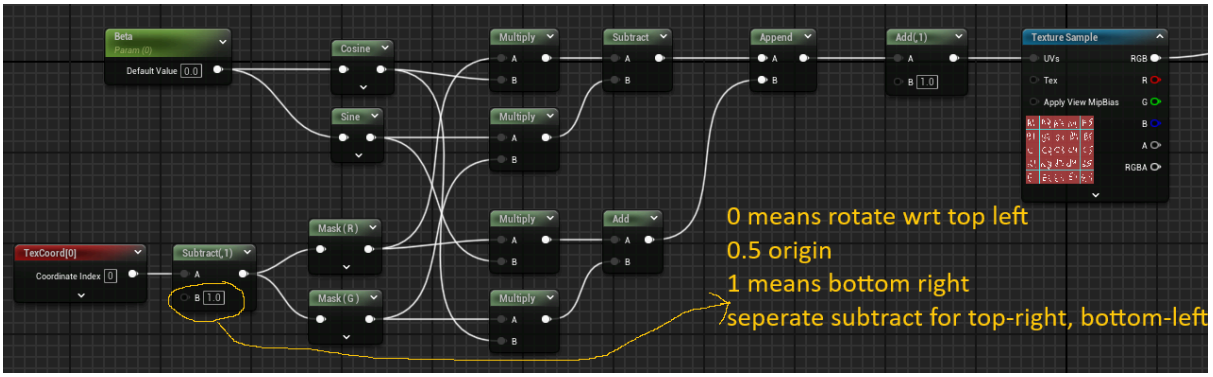
1) Adding anything in the textcoord node makes it move the texture.
2) Multiplying makes texture to scale down, as the coordinates scale the texture repeats, multiplying by 2 generally means texture repeats twice in each direction (or R/G//X/Y specifically if multiplied to them only)

- Rotate UV:



To rotate a vector with respect to origin is shown in the image. For rotation of the UV's we have to rotate each with respect to PIVOT. The PIVOT in mid can be achieved by subtracting

0.5 from both. Subtracting 0.5 will make 0.5th position UV (ie center) and then the Rotation will happen with respect to centre of the UV map instead just 0,0
The 0.5 then needs to be re-added. An example of how it's done.



- CustomRotator Node:

This node does the whole upper complex part with just one node. It rotates a vector with respect to origin.

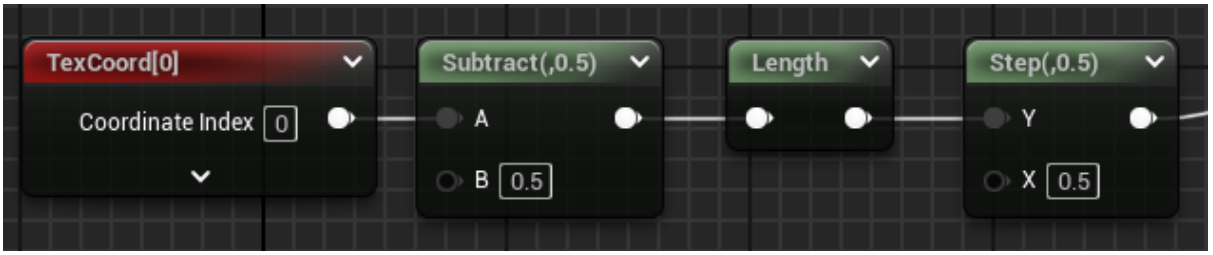
- UVs: Input vector
- Rotation Centre: 0.5,0.5 by default centre, 0,0 for top-left
- Rotation angle: 0-1, 1 means full rotation.

Drawing Shapes

Drawing Shapes:

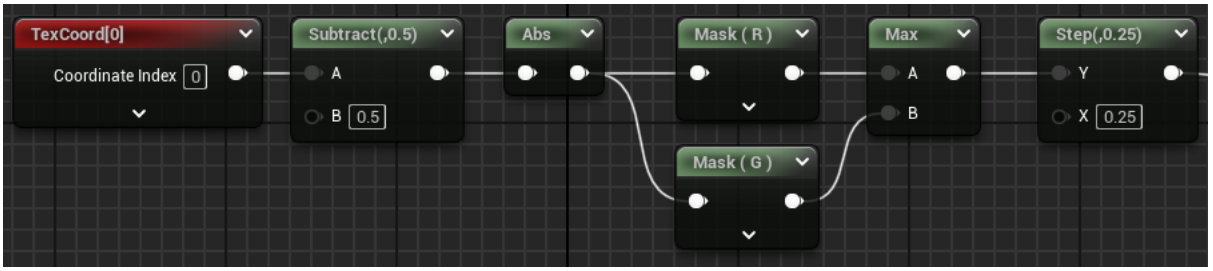
- ## Circle:

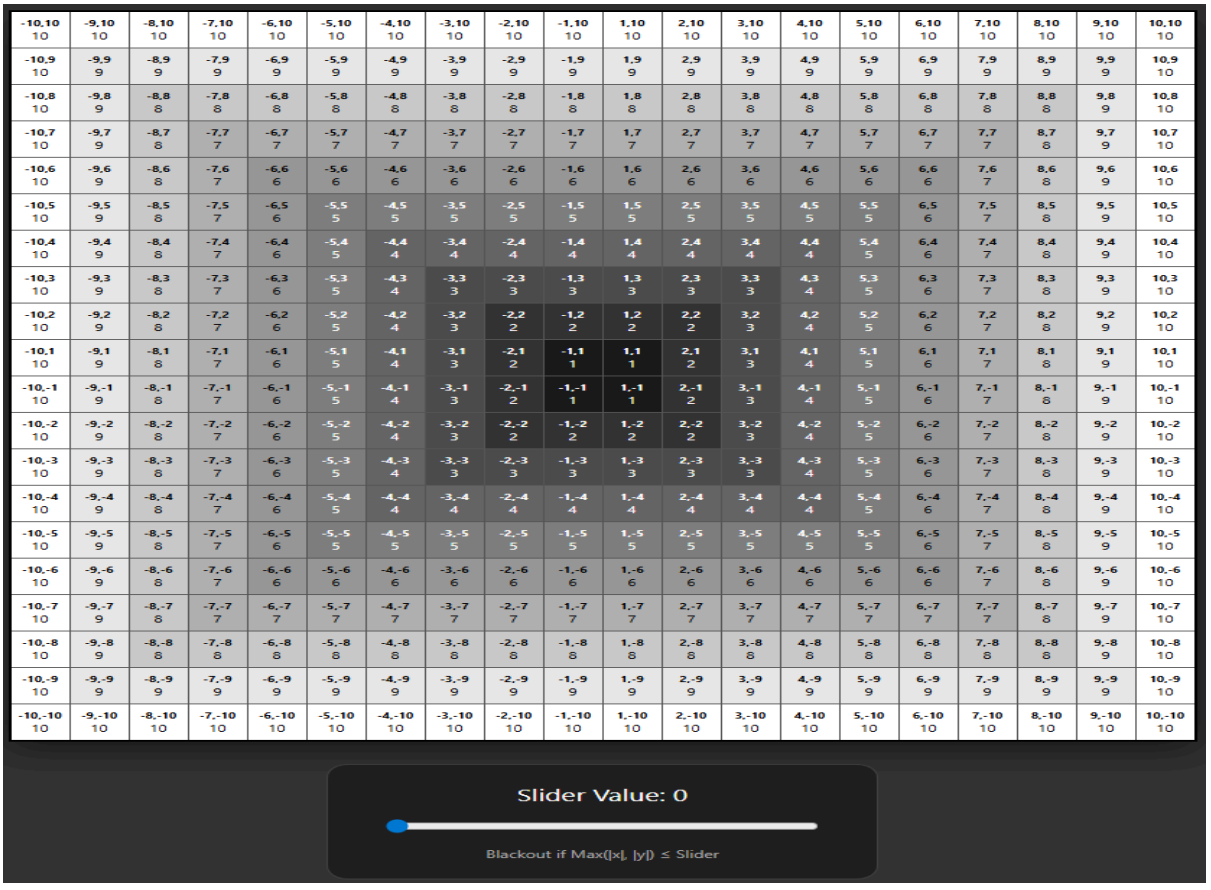
  Centre UV's then use length into step node to draw a circle.



- ## Square:

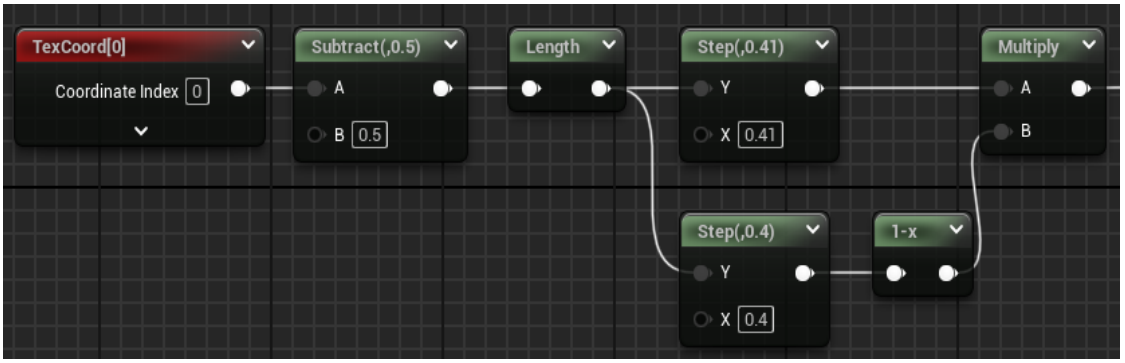
  Centre UV's and absolute them so all 4 corners are (0.5,0.5). Take the maximum out R and G and then a step node to draw squares. Maximum creates a gradient grid like Image 2. That step node then draws a square.





- Ring:

  Make a circle of x length, another of x+width radius. OneMinus x radius circle and multiply both. Multiply makes common 1’s white and others black…



- Square:

📏 Material Nodes

Material Nodes

NODE NAME	INPUTS	OUTPUTS	WORKING
Step	Y,X	Value 0/1	Outputs one if Y is greater than zero, and zero if not.
Max & Min	2	1	Returns max or min of 2 inputs…
Abs	1	1	Drops minus sign, turns negative numbers into positive
ComponentMask	1	1	Masks out R/G/B/A from a vector, or breaks a vector's dimensions
Append/ AppendMany			Adds back a vector with its RGBA components.
if	A B A>B A=B A<B	1	Takes A&B as input and returns 3rd, 4th, 5th input based on comparison of A & B. **Step is faster than if….**
sin		[-1.0, 1.0]	Outputs sin value of input, Input 1 = 2π Input ½ =π
cosin		[-1.0, 1.0]	Outputs cos value, input same as sin.
time		1	Outputs time, float increases by 1 per second
OneMinus			1-input
Saturate			Clamps number between 0-1
Clamp			Clamps number between min/max Inputs
Power			Maths power, used for contrast
Dot Product			Dot Product result of 2 vectors
Lerp			Interpolating 2 values based on alpha
Frac			Returns just decimal part of a number

- TextCoord
- CustomRotator
- SceneTexture Node:

  SceneTexture node returns basically what's drawn on the whole screen. There are different passes when rendering a frame, like base color then roughness then metallic. These are called Buffers or G-Bufferes. Each Scene Texture Id/Buffers return different passes that can be altered to make post process effects and others.

  A fun thing is that if you apply a metallic scene texture to the base color of a material and apply it on a mesh, everything you see through the mesh, it's metallic values are shown on the mesh

- SceneDepth Node (The "Background" Map)
  - What it is: A Read-Only lookup of the "Depth Buffer" (Z-Buffer).
  - When it's made: It is written during the Opaque Pass.
  - What's in it: Only the distances to Solid/Opaque objects.
  - Depth Culling: Because this pass is solid, the GPU uses it to "Cull" (delete) any solid pixels that are hidden behind other solid pixels to save performance.

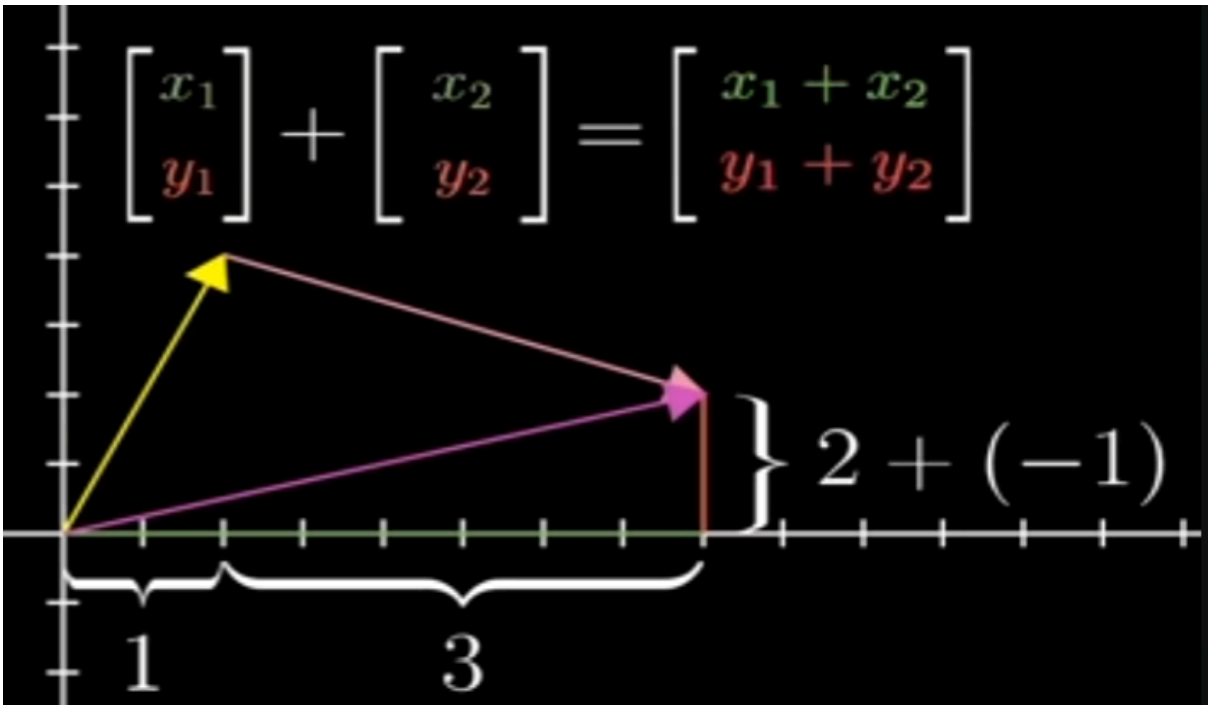
- PixelDepth Node (The "Self" Distance)
  - What it is: A mathematical calculation of "How far am I?"
  - What it returns: The distance from the camera to the specific pixel currently being processed by this material.
  - The Context:
    - If the material is Opaque, PixelDepth is the distance to the solid surface.
    - If the material is Translucent, PixelDepth is the distance to the translucent surface.
  - Key Difference: It does not know about the background. It is just a number representing the "Me" distance.

📐 Vector Maths

VECTOR MATHS

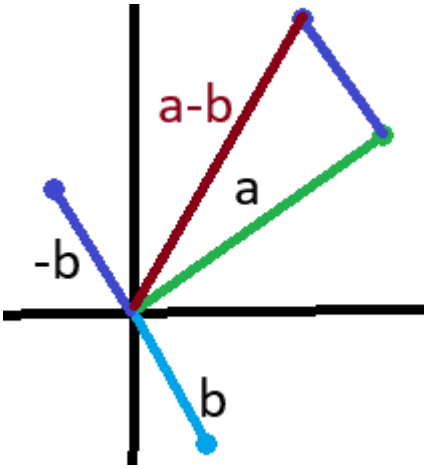
1. Addition:

A = 2x + 3y / 2i + 3j (where i & j are X&Y's unit vectors)
B = 7i + 2j
A+B = 9i +5j



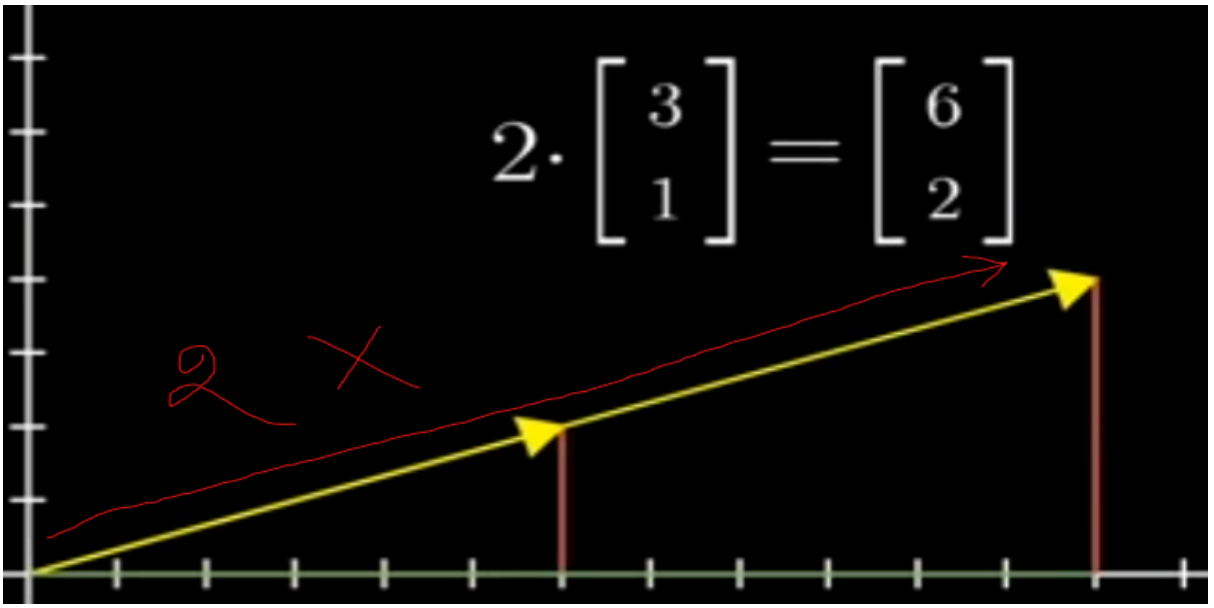
2. Substraction

A = 2x + 3y / 2i + 3j (where i & j are X&Y's unit vectors)
B = 7i + 2j
A-B = -5i +j



3. Scalar Multiplication

A = 2x + 3y
-1.5A = 3x + 4.5y



4. Dot Product

A = (2,3)
B = (3,-2)
A.B = 2x3 + 3x-2 = 0
A.B = |A|.|B|.cosΘ

A.B = B.A
If A.B = 0 -> A,B are perpendicular
If A.B = -1 -> A,B are at 180 degree
If A.B = 1 -> A,B are at zero degree
If A.B > 0 -> A,B are in same direction
If A.B < 0 -> A,B are in opposite directions

5. Cross Product

A X B = -B X A
Direction is right hand thumb rule
A X B = |A|.|B|.sinΘ .n̂

n̂ is a **unit normal vector** (a vector of length 1 pointing perpendicular to both A and B using the right-hand rule).

$$A \times B = \det \begin{pmatrix} \hat{i} & \hat{j} & \hat{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{pmatrix}$$

Niagara

NIAGARA

- ## Niagara Emitters:

  Niagara emitters as the name suggests are single emitters that can emit particles of one kind. Multiple Niagara emitters can be combined to create a good VFX effect.
  - Niagara emitters can have their own time or global time for example time of the niagara system that owns them.
  - A Niagara emitter can be an asset on its own if you want to reuse it, and it can directly sit on a Niagara system as well.
  - If an asset niagara emitter is imported into niagara systems, its modules are locked. More modules can be added and existing module properties can be edited but existing modules can not be removed by default. You can go to the settings icon in the details panel to remove the parent, meaning isolating the emitter from its parent and now all modules are fully accessible.

- ## Emitter Stack:

- ### Properties:

  1. Sim target: it is the simulation target, whether the CPU or the GPU should solve the particles. Use cpu when “you need particles to give output like where they hit”, “a few 1000 particles only”. Use gpu when “you do not require particles to calculate their hit locations, dumb particles”, “can solve millions of the particles”.
     -GPU sim particles can still receive information from blueprints, it just can’t output information.

  2. Calculate Bounds Mode: How the bounds should be calculated, if you see your particles disappearing when slightly looked away, it’s the culprit here. Fixed bounds are easy on performance.

  3. Local/World Space: Local space particles move with the emitter and do not leave a trail whereas the world space particles do so.

  4. Determinism: whether the randomness be truly random or based on a seed number provided.

  5. Interpolated Spawning: spawn particles in between if a particle is moving with a faster speed, like a bullet trail can leave dots.

6. Event handlers: Here

- ## Emitter Summery:

  It allows non vfx artists to tweak variables to get desired effects. Any variable can be set as emitter summary by right clicking on parameter and “show in summary”

- ## Execution Stages:

  ### 1. Emitter Stage

  - Spawn: Runs once when the emitter starts its loop.
  - Update: Run every single frame that the emitter is "active."
  - Example Modules:
    - Spawn Rate: "Spit out 50 particles every second."
    - Spawn Burst Instantaneous: "Spit out 10 particles right now at the start."
    - Emitter State: Decides if this specific emitter loops or runs once.

  ### 2. Particle Stage

  - Spawn: Runs once when the particle is spawned.
  - Update: Run every single frame for every single particle.
  - Example Spawn Modules:
    - Initialize Particle: Sets the starting color, size, and lifetime
    - Add Velocity
  - Example Update Modules:
    - Gravity Force
    - Solve Forces and Velocity: It actually does the math to move the particle based on the forces applied to it.

- ## Event handlers:

  Multiple event handlers can be added from the properties panel, it's not there by default.
  Example: Suppose a firework rocket goes up in the sky, when it dies we need to tell another emitter to spawn a burst of fireworks in the sky.
  So we add the Generate Death event in the rocket’s update particle stage. Click + stage at the end of properties tab and add an event handler stage in the burst emitter, here we select rocket -> Death event. This stage now works like an execution stage when the death of a particle will happen in the rocket emitter. In this stage we add a module for the same event (death in this example) “receive death event”.

  1. Death event: generated when particle dies
  2. Location event: returns location of each particle
  3. Collision event: generated when particle collides.

- Renderer:

1. Sprite Renderer:
    - Material of sprite can be changed here.
    - Particle color nodes in the material graph are used to use the particle color in the material so changing the color parameter in the niagara affects custom material.
    - The sprite pivot can be changed, which helps in rotation around the pivot.
    - SubUV: subuv is a way to cut down the material into parts, choosing 10x10 will make 100 frames of the material uv's we have. In the particle update stage, Animate SubUV module can be added to animate the SubUV frames.
    - CutOut: It's like a bounding box for each sprite, depending on how big area the sprite takes, we can decrease it by giving it a texture and choosing its alpha. For example a circle sprite will have a box bounding box by default, but if we pass its texture here and choose alpha, it will be like a circle now.
    - Bindings: We can change the color of the sprite through the color parameter (particle namespace), but if we want some other parameter to affect it we can actually directly bind that variable to the color of the sprite. Till now we were doing
      (particle namespace) color == (user namespace) MyColor
      But now we do
      Color (sprite renderer) == (user namespace) MyColor
    - Sorting: More the sorting, the higher z-order the sprite render uses and displays over the sprite renderer with a lower sorting. Just like z-order in the CSS.

2. Mesh Renderer:
    - Mesh can be changed here, multiple meshes are allowed at once.
    - Material of the meshes can be changed here as well.
    - More attributes related to mesh are in the Initialize Particles module's Mesh attributes section.

3. Ribbon Renderer:
    - Ribbon is a line between all the particles in the order they are spawned.
    - The material of this ribbon can be changed.
    - Ribbons can be in different shapes like planner, bi-planner or tube.

4. Light Renderer:

   Attaches a light with each particle, intensity and other properties can be edited.

5. Component Renderer:

   Render other particle systems, skeletal meshes and much more.

- Defaults:
  1. Emitter State Module: Sits in emitter update stage, it helps configuring emitter life cycle and the scalability. Scalability is like culling.
  2. Initialize Particles Module: Sits in the particle spawn stage, it helps give particles size, color, lifetime, position etc.
     Sprite UV mode: this setting changes the way sprite looks on screen, we can flip sprite in x or y direction with it. For example a sprite written ‘p’ can be flipped on x to look like q on y to look like b and on both to look like d.
  3. Particle State Module: when particle `Age > Lifetime` it tells the engine to kill the particle to save performance.
  4. Sprite renderer: Default renderer

- Niagara System:

The Niagara system contains multiple emitters, it can be placed in the world, its variables can be edited in blueprints.

- System Stack:

- Properties:
  1. Warmup time: Given in seconds, the system calculates for example 2 seconds of emitters and starts ahead. For example an already burning fire when the game starts at 0s.
  2. System Fixed Bounds: force all emitters inside to be fixed bounds.

- User Parameters:

  Here

- Execution Stages:

1. System Stage

- Spawn: Runs once when the effect is first created in the world.
- Update: Runs every single frame for the whole system.

- Example Modules:
    - System State
    - User Parameters
- Defaults:
    1. System State Module: Sets niagara system time, loop duration.

Parameters:

Editing a Property:

In the details panel for every module there are some properties. After every property there's a down arrow that allows us to -

1. Lerp it or set in random range or take from a curve
2. Make it new system, emitter, particle or user parameter
3. Use its value from a user parameter

Setting a Parameter:

There are 2 ways a parameter can be setted. The first one is by dragging the parameter from the parameters tab and dropping it to the suitable execution stages (where it can be set).
Another way is to add a "Set new or existing parameter directly" module in any stage. The details panel of the module will give all possible parameters that can be setted from the execution stage the module is in. In the details panel there's an option to make new ones as well.

Making a Parameter:

There are 2 ways a parameter can be created. One is by clicking the plus sign in front of the namespace (system, emitter, particle, user) in the Parameters tab.
Another one is to click the drop down in front of any property (for example lifetime in initialize particle module) and click "Read from new emitter/particle/system/user parameter"

User Parameters:

Blueprint readable.
User parameters can be made from the user parameters tab. These parameters can be edited in the details panel when dragged into the world, or through blueprints.
User parameters can be bound to properties of modules like the lifetime property of initialise particles module.

- Read/Get Value
    1. All stage modules like one in system update, or emitter spawn or particle update can read User parameters.
    2. HOW TO SET A PARAMETER
- Write/Set Value
    1. User parameters can be written from blueprints.
    2. They can be written from the details panel in the world, when a system is dragged into the scene.
    3. They can be written from the User Parameters tab in the niagara system.
    4. No stage in the system can write a user parameter.

- System Parameters:

  These parameters can be accessed from any emitter in the system. For example if we want 2 emitters to have the same spawn rate it can be made a system parameter. Click on the down arrow and 'read from a new system parameter'

    - Read/Get Value
        - System parameters can be read from all stages, particle, emitter and system
    - Write/Set Value
        - System parameters can only be set from the system update or system spawn stage.

- Emitter Parameters:

  Used anywhere in the same emitter, created the same way as the system parameter.

    - Read/Get Value
        - Emitter parameters can be read from particle, emitter stages only.
    - Write/Set Value
        - Emitter parameters can only be set from the emitter update or emitter spawn stage.

- Transient Parameters:

  Used in between while the math is done, none of our business. Gets destroyed after the math completes.

- Particle Parameters:

  When we edit the lifetime for the initialize particles module in the details panel, the engine stores the values as particle parameters. It's different for each and every particle in the emitter.
  Most of the time particle parameters are not created, but whenever created they are different for each particle.

- Read/Get Value
    - Particle parameters can be read from particle stages in the emitter it belongs to only.
- Write/Set Value
    - Particle parameters can only be set from the particle update or particle spawn stage.

- Times:

1. Timeline Duration:

   Timeline Duration is just for visualization. Duration can be changed by dragging the timeline bars.

2. Particle Lifetime:

   Particle lifetime tells how much a particle, a dot, lives after spawning.

3. System Time:

   System time tells how long a niagara system runs. For example a 5 seconds and looping infinite particle system will loop indefinitely and will reset all emitters within every 5 seconds.

4. Emitter Time:

   Different for each emitters (emitters can choose to use system time as well), emitter time tells how long an emitter stays.

Modules

NIAGARA MODULES

● Spawn Modules

Almost all spawn modules belong to the Emitter Update stage in Niagara. Spawn burst can be put into the Emitter spawn stage if in need.

1. Spawn Burst Instantaneous:

   Spawns a specific number of particles at once.

2. Spawn Rate:

   Spawns specified number of particles once per second

3. Spawn per Unit:

   Spawns particles based upon distance. Per unit (cm).

4. Spawn per Frame:

   Self explanatory…

● Velocity Module

Add Velocity: Sets particle speed and direction.
Even if the emitter is not set to Local Space, rotating the Niagara Component in the level will still rotate the velocity direction by default.
To prevent this and lock velocity direction to world space, set Rotation Coordinate Space to World/Global in the Add Velocity module details.

● Forces

Gravity, drag, vortex force and curl noise modules.

● Collision Module

Enables collision. Particles.HasCollided boolean returns true on the frame collision occurs.

● Spawn Location Modules:

1. Shape Location Module

- Spawns particles on or inside geometric shapes (Sphere, Box, Cylinder, Ring, Torus, Cone).
- Lots of tweakable parameters: surface-only spawning, axis constraints, radius distributions, and custom curve graphs for precise spawning.

2. Static Mesh Location Module

Spawns particles directly on the surfaces, vertices, or faces of a Static Mesh.

⁉️ How To's

What i made	Thought Process
Vignette postprocess material	Lerp between scene texture (post process input 0 pass) and black color, based on the centered UV's (TextCoord - 0.5) length. Add -1 to 1 for radius and CheapContrast node for hardness of the effect.
morphing particles into different shapes	-Use shape location to generate particles on one shape, use another shape location to generate on another shape. -add lerp particle attributes module in the particle update and tick particle location from the properties tab of this module, use shapelocation1 as a and shapelocation2 as b for learping. Now lerp the float over time.

GameBP

Game Blueprints

Suppose you are playing GTA5 or minecraft, we will learn everything with their example.

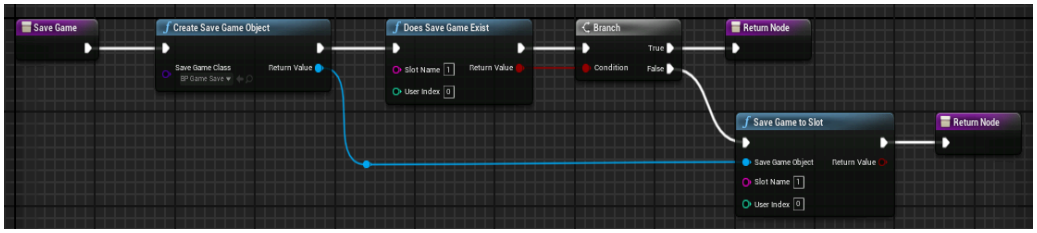
1. GameUserSettings:

When you open gta5/minecraft and if you have selected frame rate to maximum 60 last time when you exited the game, it will still be locked at 60FPS.
Saves .ini file for your game, this .ini files executes at the very first to set windowed mode, vsync and framerate cape like settings. In your options menu you can allow the player to set them directly through this class and UE handles it automatically.
It can be done through game save manually but that does not make sense.

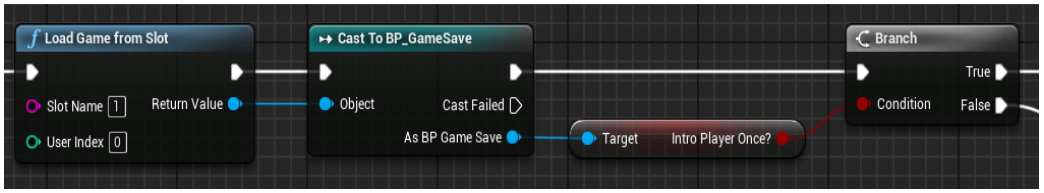
2. GameSave:

Last time you completed a mission of gta or found diamonds in minecraft, when you come back next time, they are still there.
Gamesave allows you to save variables that you can use while loading the game level next time. For replication of minecraft worlds or GTA5 save slots there are inputs called slot-name while saving the game.

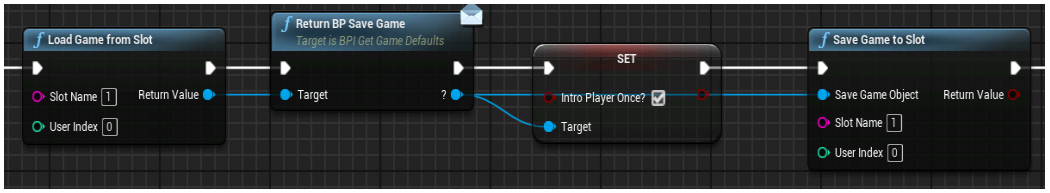
- Saving game first time:



- Getting a saves variable from a slot:



- Saving a variable to slot:



3. GameInstance:

You selected online mode in gta and wow rockstar opened story mode, this does not happen because gameinstance is there to pass variables between levels. If you have

a variable that you don’t want to save but keep it until the game is on you put it here. Gameinstance only persists through a game cycle, after its variables go to default.

4. Level:

In gta 5 there are 2 major levels, one is the main menu another one is the city where we f around. For different slots or different worlds of minecraft there are not different levels, there’s just one WORLD level notch made and he brings variables from save game to make it look like the game you left last time.
Levels are hard to explain but as you progress you know what it is.

5. Animation Sequences:

These are created through sequencer and the timeline window. These are cut scenes of a gta game.

6. GameMode:

This is what tells a level that when a player clicks play, which character to use, which controller to use, which hud class (UI) to use. It's setted through the world settings tab.

7. PlayerController:

You have a superhero game where you can become iron man and spider man. The player controller defines what’s in common, usually we put the logic of looking around in the level with mouse input, as both spidy and tony can look around. Apart from that we put pause menus, inventory (if both share the same inventory) and showing mouse cursor or hiding it when we’re navigating through UI or playing the game.
For a multiplayer game every player playing only has one controller, for a co-op game every physical player playing has exactly one controller. For a real human being playing a level there’s exactly one Player Controller.

8. Pawn/Character:

A level can be controlled through a pawn or a character, character is a child blueprint of pawn with more modules such as Movements.
We design 2 different characters for iron-man and spider man as they both have different sorts of gameplay. Gta’s michel franklin and travor are children of the same character blueprint, this line you might not be able to digest yet if you’re new in ue5. But let me clear, gta5’s all three character walk drive shoot the same way, making 3 different character classes would be a waste of time and memory. Instead you created one master character that can walk, shoot, talk, drive and made children of this blueprint so that you can give them a different body (skeletal mesh), different voice, and different superpowers (like slowing the time or so).

9. HUD class:

I don’t use it, it basically allows you to build that UI of map, gun switching and the main hud we see while playing the game. We usually do it with character and controller class.

10. GameState:

In an online game you see some variables are common for everyone, like players alive of PUBG/FORTNITE. It’s here, we put all the logic of global rules like in PUBG new state you can make an enemy your teammate after knocking him out, that logic relies here, all the teams logic and so on.

11. PlayerState:

The simplest explanation is if you have kept inventory turned off in minecraft when you die, inventory items drop in the world, which you do with player state. If keep inventory is on then you save inventory in game save. Although in real minecraft in both cases inventory is saved in gamesave. In the player state you put variables that you want to set default when the player dies.

12. Spectator Class:

In fornite and pubg when waiting for revival you can roam around freely in the world, no collision nothing whatsoever, just a flying camera and controls. You put logic like the spectator class should not be able to see/hear other than his teammates as in case they can tell the location of enemies through flying to their teammates.