BigWorld Technology

Client Training

Outline

- Configuring the Client
- Entities
- Models
- Animations and Actions
- Cameras and Viewing the World
- 3D Engine
- Gaming Aspects
- Particle Systems and Effects
- GUI
- Job System
- Profiling and Engine Statistics

Session 1
Configuring the Client

Configuring the Client - Overview

- Configuration files
- Personality Script
- Connecting to Server

Configuration files

- XML based configuration files
- Engine config:
  - Configure the (C++) client engine
  - ***<resources_folder>/engine_config.xml***
- Script config:
  - Used to configure scripts on startup
  - ***<resources_folder>/scripts_config.xml***
- User Preferences:
  - Configure graphics, display and game specific settings
  - ***<working_folder>/preferences.xml***

Configuration files

- Entities:
  - List all the entities to be used
  - ***<resources_folder>/entities.xml***
- Resources:
  - Specify resources to be used in the game
    - e.g. the loading screen
  - Override these for your own purposes
  - ***<resources_folder>/resources.xml***

Personality Script

- Global script used to start up the client
- Load and Update settings
- Initialise the game
- Handles the global Logic
  - In game menus
  - Connection to Server
- There are personality scripts for the base and cell apps as well
- Location:
  - ***<resources_folder>/scripts/client/<script_name>.py***

Personality Script

- Interaction is handled via Notification Methods or Callbacks

Method	Description
init(…)	Called when BigWorld is ready to initialise. *(engine, script and preference config files are passed as parameters)*
fini(…)	Called when BigWorld shuts down
handleMouseEvent(…)	… or moves the mouse
onChangeEnvironments(…)	Called when the player goes from outside to inside or vice versa

Connecting to Server

- Connecting to Server is done via a BigWorld python module method
- ***BigWorld.connect(hostName, args, connectionCallback)***
  - ***hostName***: String with the IP or domain address of the login server
  - ***args***: Optional Python object with
    - ***username***: Name of player will display and be represented by
    - ***password***: Password to ensure player authenticity
    - ***inactivityTimeout***: Seconds before player is disconnected if no updates are received
  - ***connectionCallback***: Optional Python method that will be called with the various stages of the login process

Connection Callback Function

- ***def connectionCallback(self, stage, status, msg)***
  - ***stage:*** Defines the current phase of connection
    - **0** Client could not initiate login
    - **1** Client has connected to Login Server
    - **2** Client has received data from Login Server (connection is complete)
    - **6** Client has been disconnected from Login Server
  - ***status:*** Indicates outcome of that phase
    - 'NOT_SET' : Not set
    - 'LOGGED_ON' : Account Login succeeded
    - 'CONNECTION_FAILED' : Login failed: Unable to contact login server
    - etc
  - ***msg:*** Text message describing the response

Connection Information

- ***BigWorld.LatencyInfo().value***
  - A read only Vector4 of this clients network latency information
  - ***isOnline, minLatency, maxLentency, avgLatency***

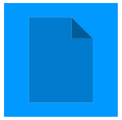
- ***BigWorld.serverDiscovery***
  - ***servers:*** List of available servers as serverDiscovery::Detail objects
  - ***searching:*** 1-Initiates search of BigWorld servers on LAN

    0-Stops current search and clears server attributes
  - ***changeNotifier:*** Python function to be called every time a Details object is created or updated in the servers attribute list

Connection Information

- ***serverDiscovery::Details*** ***(read-only)***
  - ***hostName:*** Name of host machine
  - ***ip:*** IP address of host machine, as integer. 1st byte is most significant
  - ***port:*** Access port that the host machine is listening to for connections
  - ***uid:*** User ID of host to uniquely identify multiple servers on a machine
  - ***ownerName:*** Name of the user who launched server
  - ***usersCount:*** Total connection attempts made by any client to this host
  - ***universeName:*** Name of universe currently playing on server
  - ***spaceName:*** Name of space currently playing on server

Demo Time

- Presented:
  - Configuration XML Files
  - Personality Script
  - Server Connection
  - Demo App: 

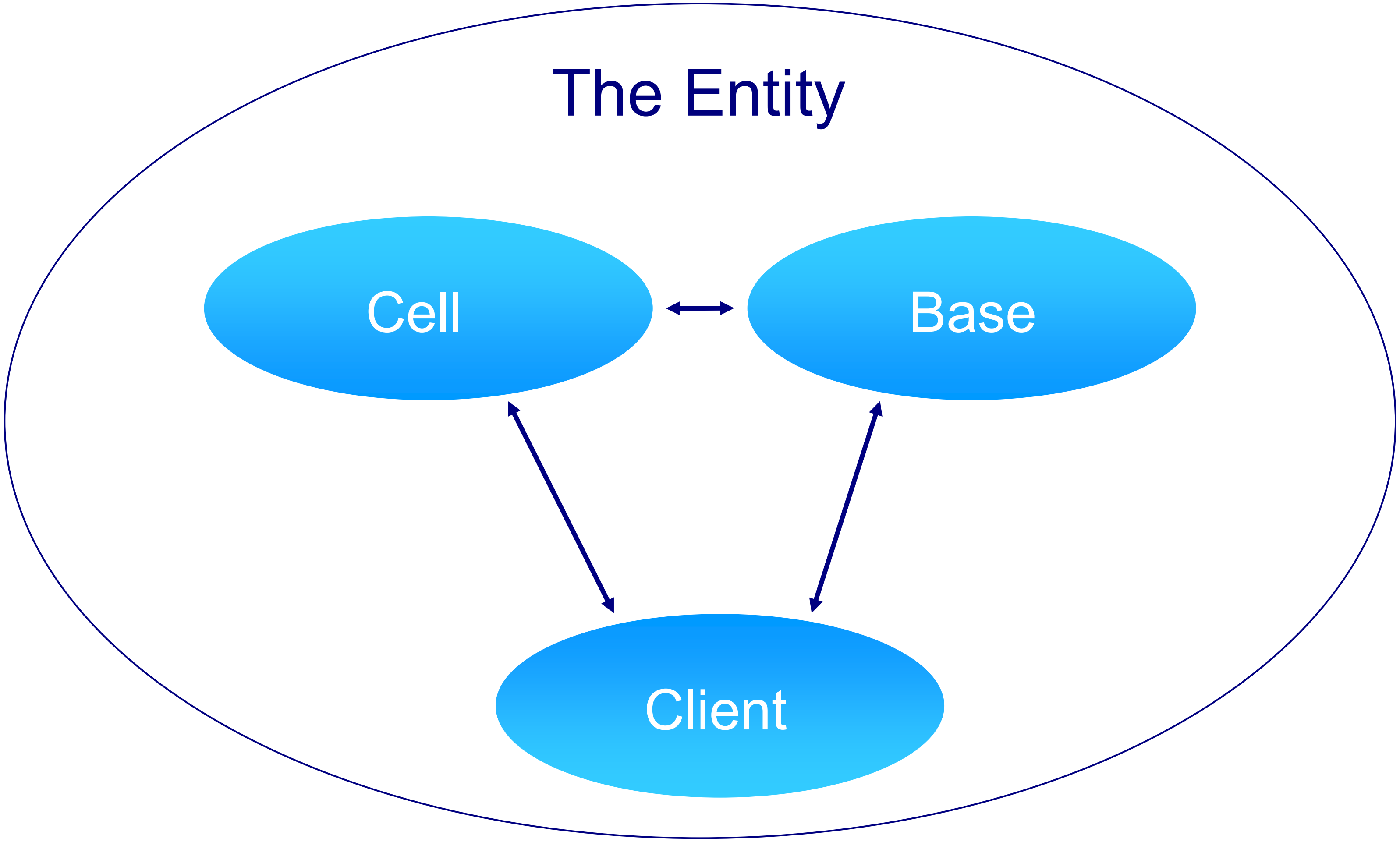
bigWORLD™ TECHNOLOGY

Session 2
Entities

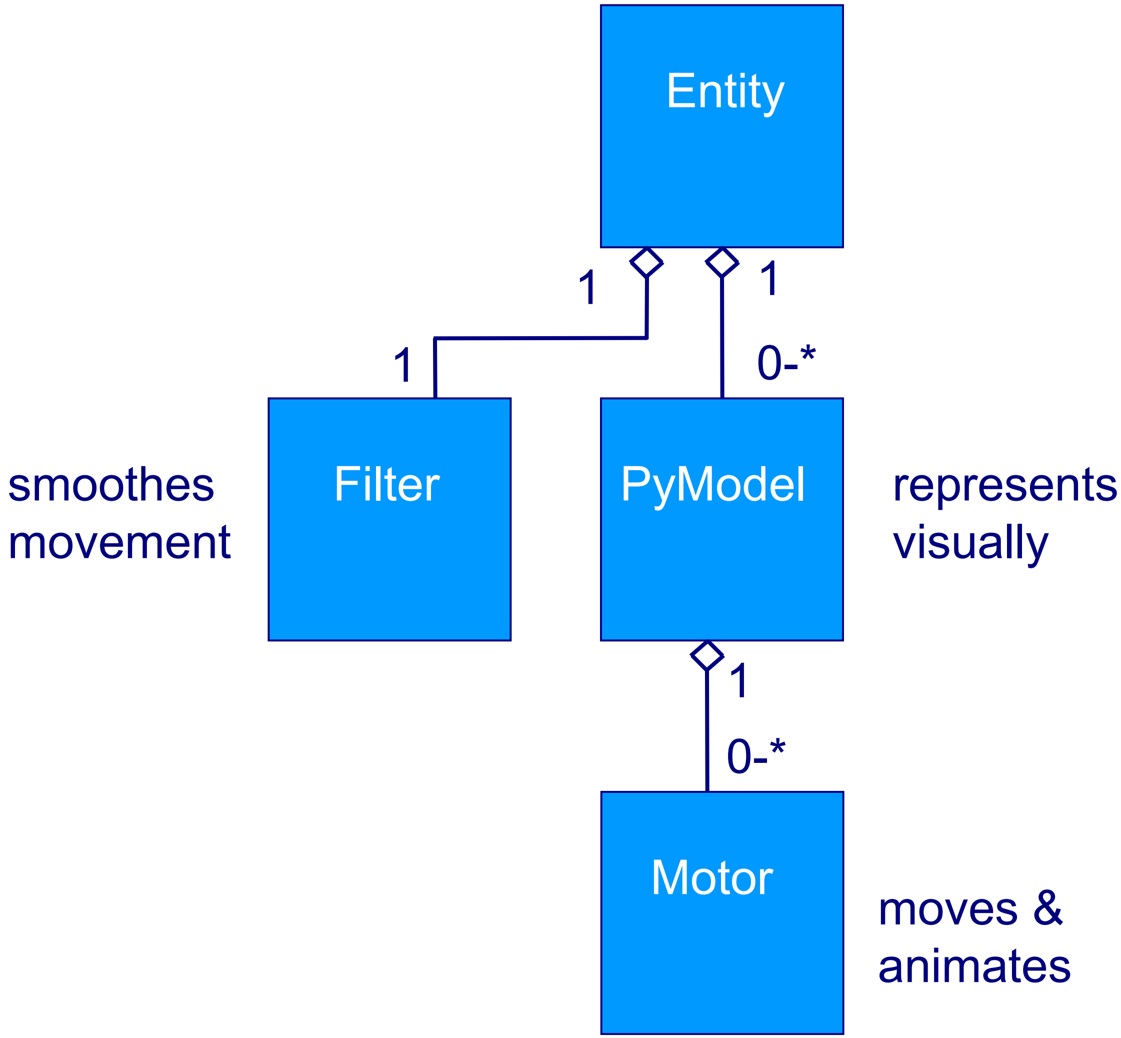
Entities - Overview

- Entity Architecture
- Client Entity
- Entity Script
- Client – Server Communication
- Physics
- Movement Filters
- Player Entity

Entity Architecture



Client Entity



Entity Script

Recall Distributed Entity Model from Server Training

- Entities can exist on Cell, Base and Client
- Interface (definition), Python (implementation)
- The script implements:
  - ***<ClientMethods>***, as described in the definition file
  - Property notification methods:
    - ***set_<propertyName>() # From self.health = 20***
    - ***setNested_<propertyName>() # From self.myFixedDict["x"] = 2***
    - ***setSlice_<propertyName>() # From self.myArray.append(7)***
  - General notification methods
  - Any other client-side behaviour

Entity Notification Methods

Method	Description
onEnterWorld(…)	Called when BigWorld has inserted the Entity into the game world
targetFocus(…)	We have acquired a new target
targetModelChanged(…)	The target's model has changed
onControlled(…)	Lets us know our *controlled* state has changed

Client – Server Communication

- Client entities have base and cell mailboxes to communicate to the server
- ***Entity.base.methodName()***
  - Calls method on the base component of entity
  - Only valid for the player entity
- ***Entity.cell.methodName()***
  - Calls method on the cell component of entity
  - Valid for all entities on the client with a cell component

Physics

- Entity physics is used to manipulate the movement of the entity
- Only controlled entities are allowed to use physics
- Each entity must initialise its physics attribute with a physics type
- Example:
  - ***self.physics.type = BigWorld.STANDARD_PHYSICS***
    - Set physics to standard player-style physics
  - ***self.physics.collide = True***
    - Activates collision detection

Type	Description
0 – Standard	Falls, collides with scenery
2 - Limpet	Use **physics.chase** to stick to an entity

bigWORLD TECHNOLOGY

Using Physics

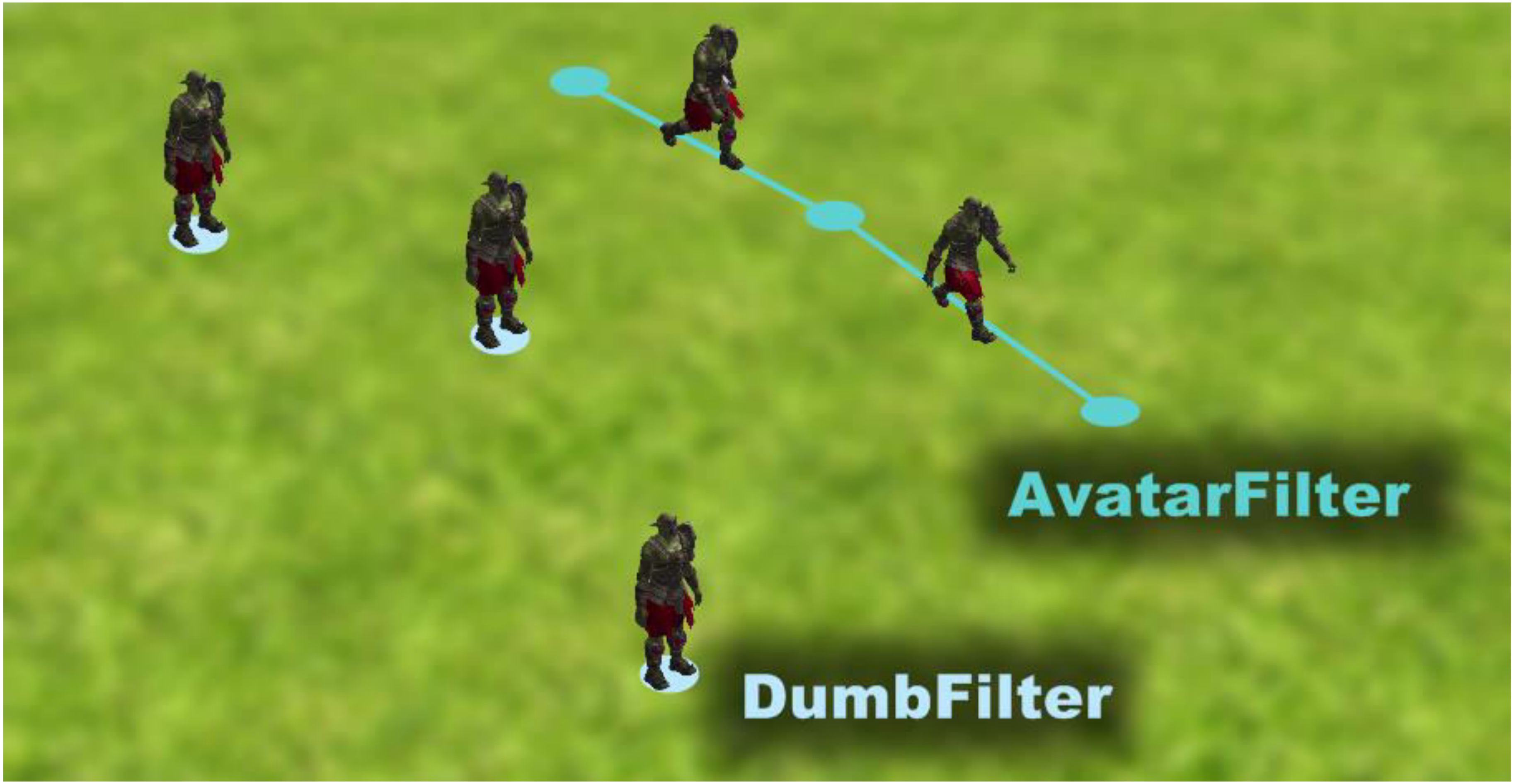
Method	Description
seek()	Make the player travel to a position in the world

Attribute	Description
velocity	Set the velocity of the player
dcLocked	Temporarily detach Entity yaw from mouse control
fall	Set to true to allow the entity to fall

Movement Filters

- Filters allow for smooth movement in the game
- Position updates from the server may be infrequent
- Use an Entity Filter to provide a real-time position
- Example:
  - ***`Entity.filter = BigWorld.AvatarFilter()`***
  - Creates a simple filter and attaches it to the entity
  - Linearly Interpolates the positions received from the server

Movement Filters

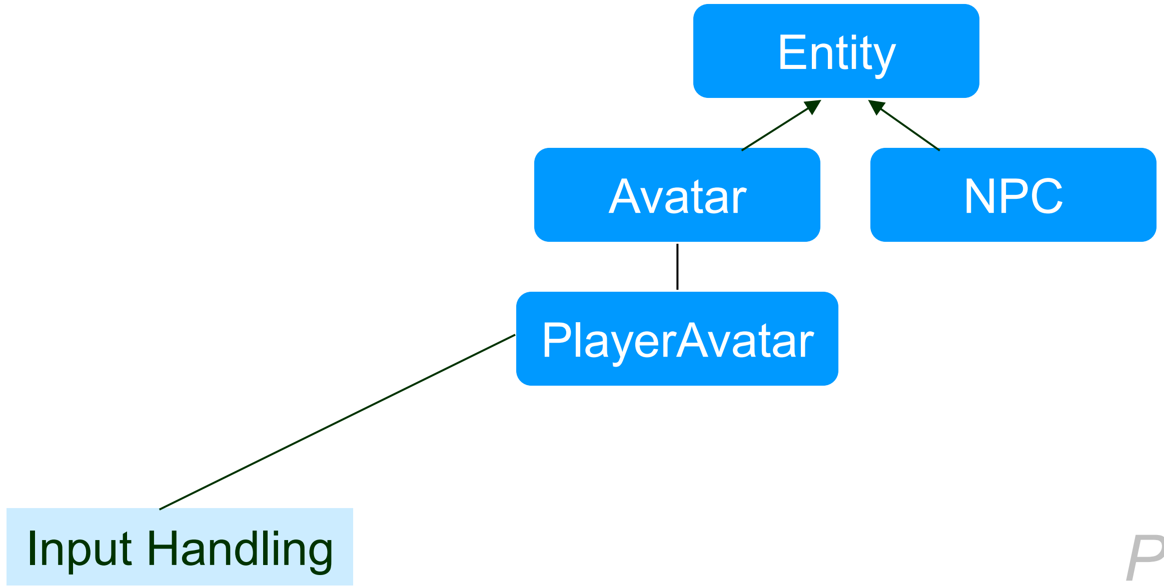


bigWORLD TECHNOLOGY

Movement Filters

Filter	Description
DumbFilter	Does nothing
AvatarDropFilter	As above, but also drops the entity onto the ground
BoidsFilter	Control many models using a flocking algorithm

Player Entity



Player Controlled

Player Entity

- The player entity's Python object must:
- Be a class that inherits from an actual Entity class
- Be called ***Player<ParentClassName>***
  - Example: ***PlayerAvatar***
- ***BigWorld.player()***
  - Returns current player entity
- ***BigWorld.player(entity)***
  - Sets new player entity
  - No effect if no ***Player<entity>*** class exists
  - ***onBecomeNonPlayer()*** called on the old player entity
  - ***onBecomePlayer()*** called on the new player entity
  - There can only be one player entity at any one time

Controlled Entities

- A Controlled Entity is directed by a client, and sends position updates to the server (e.g. Player)
- A Non-controlled entity receive server updates
- Any number of entities can be controlled, *i.e.*, affected by user input
  - Player in big boat
  - A "Redeemer" weapon
  - A "Lost Vikings" game

Demo Time

- Presented:
  - Entities
  - Physics
  - Filters
  - Player
  - Demo App:

Session 3
Models

Models - Overview

- Model Overview
- Model Creation
- PyModel
- PyModelNode
- PyAttachment
- PyFashion
- Model LOD
- Motors
- Trackers

Models - Overview

- The Model represents a 3D Object
- Entities may have 0 or more models, usually 1
- They have one primary model, and *n* secondary models
- They can be attached to other models too, more about that later…
- Examples :
  - ***pyModel = BigWorld.Model(modelFile,…)***
    - modelFile takes **`.model`** paths and returns a **`PyModel`** object
    - Can pass any number of **`.model`** paths, which are joined to produce a single **`PyModel`** according to node structures
  - ***pyModel = BigWorld.PyModelObstacle(modelFile,…)***
    - Create a model that exists in the collision scene

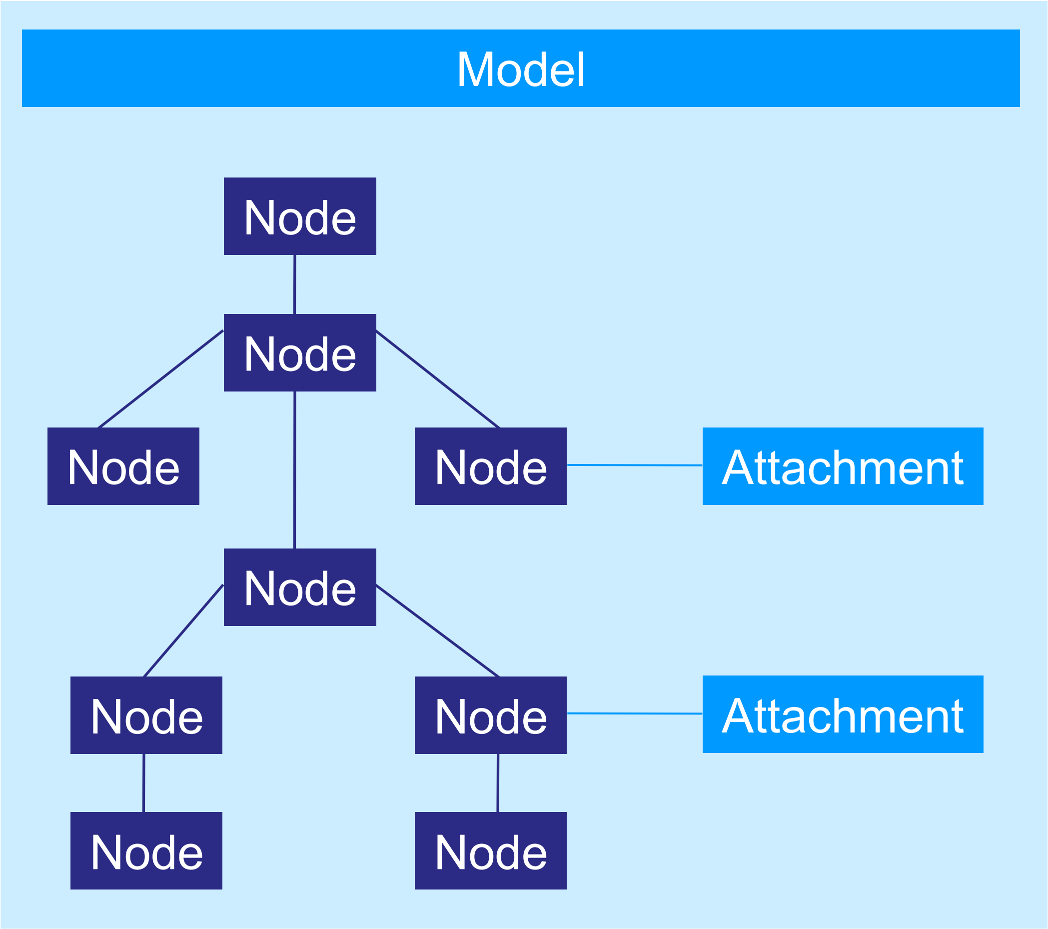
Models - Overview

- ***entity.model = pyModel***
  - Sets the main model for the entity
- ***entity.addModel(pyModel)***
  - Adds secondary models to the entity (*e.g.*, a spell effect)
- ***entity.delModel(pyModel)***
  - Removes secondary models from the entity
- ***entity.models***
  - Python list of secondary models
- ***model.position = (x,y,z)***
  - Set the position of the model

Model Creation

- Models and animations are created in 3ds Max and exported with the BigWorld exporter
  - ***.primitives*** Raw mesh data
  - ***.animation*** Raw key frame animation data
  - ***.visual*** Node hierarchy, rendering info, bounding box.
    If a node name starts with **HP_**, then it is a hard point attribute that returns a ***PyModelNode***. See example file.
  - ***.model*** Refers to the **.visual** file, and includes animations, actions, dyes.
    See example file.

PyModel



Model
Node
Node
Node
Node
Attachment
Node
Node
Node
Attachment
Node
Node

PyModelNode

- Use nodes to make attachments to your model
- Two ways to access a ***PyModelNode***:
  - ***model.HardPointName*** attributes
    - Returns the node called ***HP_HardPointName***
  - ***model.node("nodeName")*** method
    - Returns the node called **nodeName**
- ***model.myHardPoint = pyModel***
  - Attaches a secondary model to a node on main model (*e.g.*, a gun)
- ***node.attach(PyAttachment)***
- ***node.detach(PyAttachment)***
- ***node.attachments***

PyAttachment

- Any number of ***PyAttachment*** objects can be attached to any ***PyModelNode***
  - …but a ***PyAttachment*** can be attached to only one **PyModelNode** at any one time
- ***PyAttachments*** include:
  - ***PyModel*** Any ***PyModel*** can be attached to another
  - ***ParticleSystem*** Describes particle behaviour
  - ***PySplodge*** A generic shadow cast by the sun
    - Width, height and LOD can be customised
    - Shadow texture configured in ***engine_config.xml***
  - ***GuiAttachment*** Allows attachment of GUI components

PyFashion

- ***PyFashion*** objects alter the look of a ***PyModel***
- ***PyDye(matter, tint)***
  - Changes the material of parts of a ***PyModel*** as specified in the **.model** file. See python_client.chm.

Model LOD

- Model Level of Detail can be used to increase performance.
- Models can define simplified versions of complex models.
- BigWorld automatically swaps the models based on the cameras position.
- Controlling LOD by:
  - Material: define fewer shaders.
  - Mesh: create models with fewer polygons.
  - Node: remove node influences in skeletons.

Motors

- Motors are used to move, orient and animate models
- Entity Models by default are drawn at the origin
- When a ***PyModel*** is assigned to ***Entity.model***, it has an ***ActionMatcher*** motor added to it automatically
- The action matcher moves a model to where the entity is
- It also performs animations – we'll talk about this later

Motors

- Examples:
  - ***motor*** **=** ***BigWorld.Propellor(parameters)***
    - Creates a ***propellor*** motor
  - ***self.model.addMotor(motor)***
    - Adds ***motor*** to the model
    - Multiple motors will each get a turn to impact the **PyModel**, which may cause clashes
  - ***self.model.delMotor(motor)***
    - Removes ***motor*** from the model

Motor Types

Motor	Description
ActionMatcher	Moves PyModel to to where the Entity is, and shows an appropriate animation
Oscillator	Rotate model back-and-forth (security camera)
LinearHomer	Travel directly to target
Bouncer	Use physics to bounce model (grenade)

Trackers

- Trackers override the animation for a node
- Changes yaw and pitch of a ***pointingNode*** based on a direction provider
- Blends nodes connected to the ***pointingNode*** to in-between positions. Blending determined by ***TrackerNodeInfo***
  - Example: Turning head to face another entity, partially turning neck and shoulders
- They point a node(s) in a direction, after the animation has been applied

Trackers

- Example:
  - `tracker = BigWorld.Tracker()`
  - ```
    tracker.directionProvider =
        BigWorld.DiffDirProvider(sourceMatrix,
               targetMatrix)
    ```
  - ```
    tracker.nodeInfo =
        BigWorld.TrackerNodeInfo(self.model,
           "Head", [( "Neck", 0.5 )],
        "None", -100, 100, -100, 100, 100 )
    ```
- Attach and remove Tracker:
  - `self.model.tracker = tracker`
  - `del self.model.tracker`

bigWORLD™ TECHNOLOGY

Demo Time

- Presented:
  - Model Creation
  - PyModel and PyModelNode
  - Attachments
  - Fashions
  - LOD
  - Motors
  - Trackers
  - Demo App:

Session 4
Animations and Actions

Animations and Actions - Overview

- Playing Actions
- Actions Architecture
- ActionQueuer
- ActionMatcher

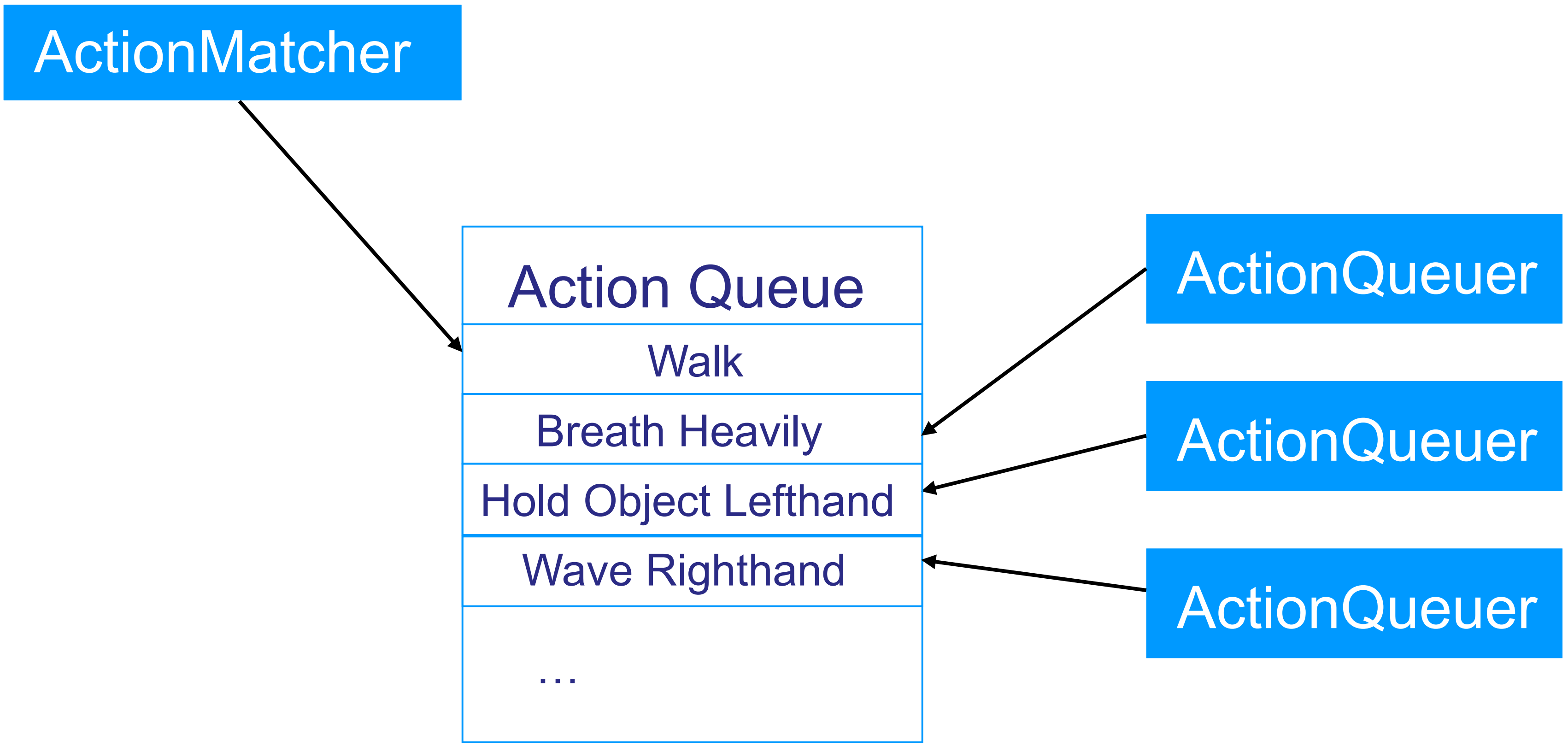
Animations and Actions - Overview

- Animations perform the movements on the 3D model using skeletal animation or morphing
- Actions are used to play animations on models but have additional properties

Playing Actions

- **`ActionQueuers`** are used to control animations from Python
- Play an action immediately:
  - **`Model.Jump()`**
- Play an action in 1 second:
  - **`Model.Jump(1.0)`**
- Play an action in 1 second, then call back the script when complete:
  - **`Model.Jump(1.0, self.onJumpComplete)`**
- Play an action, then play another action when that is finished.
  - **`Model.Jump().Land()`**

Actions Architecture



ActionQueuer

- Store an **`Action`** to call at anytime:
  - **`myJumpAction = Model.action("Jump")`**
  - **`myJumpAction = Model.Jump`**
- <u>**`myJumpAction(afterTime, callBack, promoteMotion)`**</u>
  - Adds the **`ActionQueuer`** to the **`ActionQueue`**
  - All arguments are optional
    - **`afterTime`** - Delay before queuing the action
    - **`callBack`** - Method to call when action completes
    - **`promoteMotion`** - Boolean defining whether animation will move the entity

ActionQueuer

- All Actions are blended
  - No popping of animations
  - Blend in / Blend out
  - Blend between other actions that are playing
  - All on a per-node basis

ActionQueuer Attributes

Attribute	Description
track	Integer. Actions will replace any action playing on the same track. -1 means 'no track'
blendOutTime	Time in seconds for an action to completely stop affecting the model
displacement	Distance in metres the action moves the model

ActionMatcher

- Chooses and displays animations, given only the basic movement updates from the server
  - Added as a default motor when a **`PyModel`** is assigned to **`Entity.model`**
- Every action can have a **`<match>`** section, which can contain a **`<trigger>`** and a **`<cancel>`** section
  - Note matches like minSpeed, maxSpeed
  - See example here : base.model
  - This is setup in ModelViewer
- See the **`ActionMatcher`** at work:
  - **`BigWorld.debugAQ(BigWorld.player())`**
  - **`Tests.ActionMatcher.test()`**

bigWORLD TECHNOLOGY™

ActionMatcher Attributes

- **`matcherCoupled`**
  - Defaults to 1. Set to 0 to turn off
- **`inheritOnRecouple`**
  - Defaults to 1. Player Entity will be moved to **`PyModel`** position to prevent visual jarring
- **`lastMatch`**
  - Name of the last action taken
- **`matchCaps`**
  - List of numbers between 0 and 31 that represent the a user-defined state of the **`ActionMatcher`**
  - Actions can specify certain states in order to be selected for matching

ActionMatcher Attributes

- **`turnModelToEntity`**
  - Determines if **`ActionMatcher`** should adjust **`PyModel`** yaw to match direction of entity
    - If set to 0, ActionMatcher will not change yaw at all
- **`footTwistSpeed`**
  - Rate at which the entire **`PyModel`** can twist yaw to face the current direction of its entity owner

Demo Time

- Presented:
  - Animations
  - Actions
  - ActionQueuer
  - ActionMatcher
  - Demo App:

Session 5
Cameras and Viewing the World

Cameras and Viewing the World - Overview

- Camera Types
- Creating and Using Cameras
- Setting up views

Camera Types

- CursorCamera
  - Uses **`DirectionCursor`** to follow line of sight at a desired distance from the entity
- FreeCamera
  - No limits, no collision, freely flys around the world, controlled by arrow keys and mouse
  - **`freeCamera.fixed = True`** will stop any movement

Creating and Using Cameras

- **`BigWorld.CursorCamera()`**
  - Creates a **`CursorCamera`**
- **`camera.target = BigWorld.PlayerMatrix()`**
  - **`BigWorld.PlayerMatrix()`** returns **`MatrixProvider`**
  - **`target`** is updated as player position is updated
- **`BigWorld.camera(`*`camera`*`)`**
  - Only one camera can be in use at any one time
- **`BigWorld.firstPerson(true)`**
  - Cursor camera operate in 1st or 3rd person mode

Creating and Using Cameras

- **`camera.set(`*`transformMatrix`*`)`**
  - Sets the world transform of the camera
- **`camera.position`**
  - Current position of the camera
- **`camera.direction`**
  - Current orientation of the camera
- **`camera.matrix`**
  - Current world → camera transform
- **`camera.invViewMatrix`**
  - Current camera → world transform

Setting up view

- **BigWorld.projection()**
- Returns the singleton `projection` object
- Contains:
  - **`nearPlane`**
    - Distance to the near clipping plane
  - **`farPlane`**
    - Distance to the far clipping plane
  - **`fov`**
    - Field of view, between 0 and pi
  - **`rampFov(`*`newFOV, timeAllowed`*`)`**
    - Changes to the given field of view over the specified time period

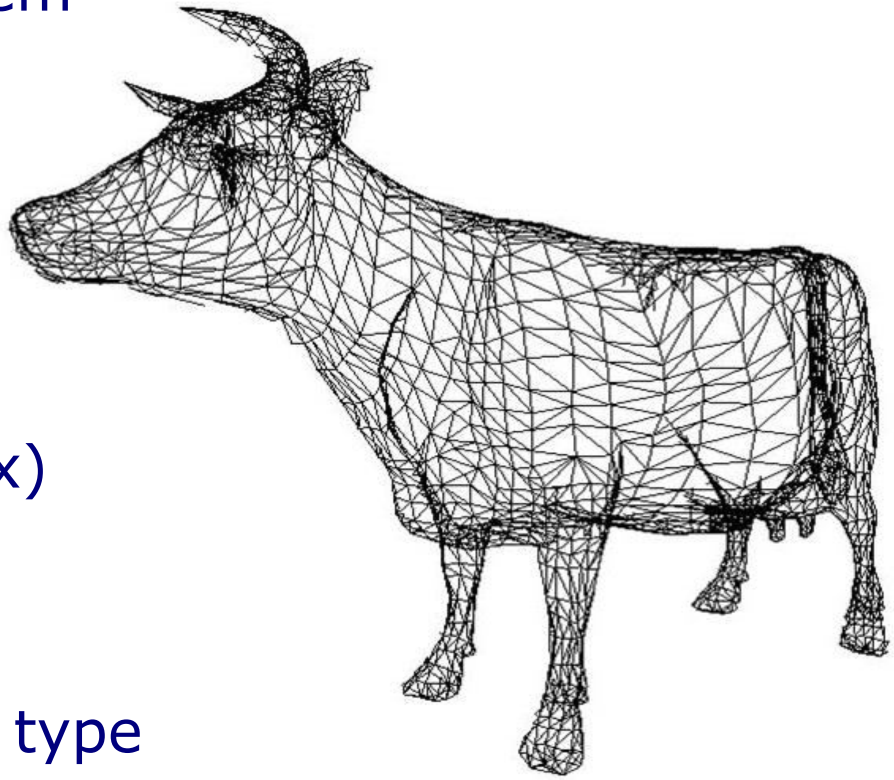
Session 6
3D Engine

3D Engine - Overview

- Engine Overview
- Visuals
- EffectMaterials
- Lighting
- Textures
- Height-Mapped Terrain

3D Engine - Overview

- DirectX9 engine
- Left-handed coordinate system
  - X–right, Y–up, Z–into the screen
- Extensive use of `.fx` files
- Height-mapped terrain
- Skeletal animation
- Skinning (3-bones per vertex)
- Render channels for delayed rendering
- Called Moo because quick to type



Visual

- Low-level renderable object, exported from content creation application
  - 3ds Max, Maya
- Most in-game objects rendered with visual
  - Characters, scenery objects, shells, etc...
- Contains:
  - Primitives   Vertices and indices
  - Skeleton   Nodes hierarchy
  - Materials   **`EffectMaterial`**
  - Collision data   BSP tree

Effect Material

- Uses ID3DXEffect
- Variables
  - Artist-editable variables
  - Auto variables (using variable semantics)
- Technique annotations
  - Informs the renderer about skinning, normal mapping, render channels and graphics settings.

Lighting

- 8 Dynamic lights
  - 2 Directional lights
  - 4 Point lights
  - 2 Spot lights
  - Culled to objects
- Static vertex lighting
  - Shells and indoor objects

Property	Directional	Point	Spot
Colour	✓	✓	✓
Direction	✓	✗	✓
Position	✗	✓	✓
Inner radius	✗	✓	✓
Outer radius	✗	✓	✓
Cone angle	✗	✗	✓

Textures

- The engine compresses and scales textures based on rules defined by **`TextureDetailLevel`** or a **`.texformat`** file
- **`TextureDetailLevel`** matches the resource name – or part of it – to a rule, in order to decide:
  - Texture format
  - Texture format when compressed
  - Whether the texture needs to be rescaled
- Behivour in relation to texture quality settings **`.texformat`** is an XML file that can be used to determine the format of a texture.
  - If a **`.texformat`** file exists with the same name as a texture, then the file will decide the format of the texture

Textures

- Compressed and scaled textures are cached to **`.dds`** (or **`.c.dds`** if compressed) files with the same name as the source texture
  Mip-map chain generation rules

Height-Mapped Terrain

- 100x100 metre blocks
  - Same size as chunks
- Height pole frequency defined per space
- Arbitrary number of texture layers
  - 4 layers per pass
  - Layer mask detail set per space
  - Independent of height pole frequency
- Alpha component contains specular luminance

Demo Time

- Presented:
  - Effect Files
  - Lighting
  - Textures
  - Demo App:

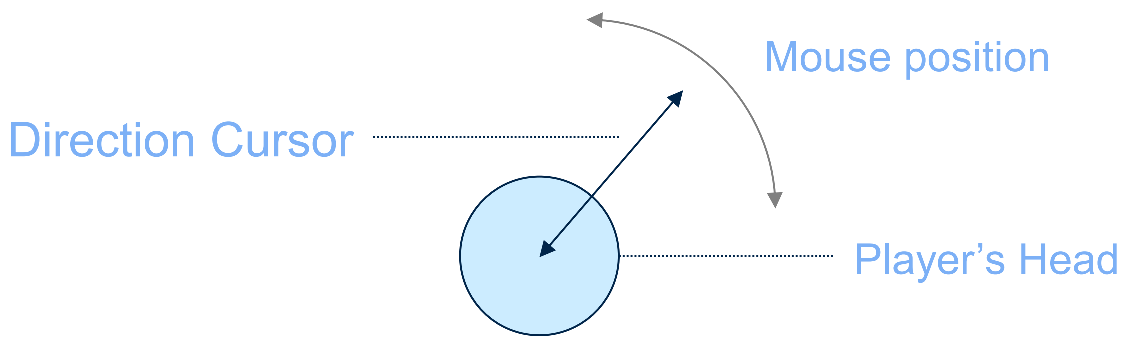
Session 7
Gaming Aspects

Gaming Aspects - Overview

- Direction Cursor
- Targeting
- Traps
- Matrix and Vector4 Providers
- Profiling & Engine Statistics

Direction Cursor

- The Direction Cursor is a vector that extends from the Player's head
- Mouse input affects the angle of this vector to produce a potential line of sight
- Provides a full, dynamic matrix (position and orientation)



bigWORLD TECHNOLOGY

Direction Cursor

- `dc = BigWorld.dcursor()`
  - Get the Direction Cursor object
- `BigWorld.dcursor(dc)`
  - Activates the direction cursos (activated by default)
- Many BigWorld objects can use the Direction Cursor
  - Trackers can animate body parts to face along line of sight
  - ActionMatcher animates PyModel to match pitch/yaw
  - Cursor Camera sits behind head, looking along line of sight

Targeting

- **`BigWorld.target`**
  - Access the BigWorld Targeting System
- **`BigWorld.target()`**
  - Retrieves the Entity closest to the center of the screen
  - Or you can use your own view vector for targetting
- Use Bit Fields to allow selective targetting
  - **`BigWorld.target.caps( [NPCs, Monsters] )`**

Traps

- Triggers a callback whenever the player entity on a given client enters or leaves the trap
- **`BigWorld.addPot(`*`centre, radius, callback`*`)`**
  - Returns a trap ID that can be saved
  - Creates a spherical, player-only trap denoted by the given **`MatrixProvider`** and ***`radius`***, invoking ***`callback`*** when triggered
- **`def trapCallback(`*`entered, trapID`*`)`**
  - ***`entered`*** will be **`1`** if player has entered the trap, **`0`** if left it
- **`BigWorld.delPot(`*`trapID`*`)`**
  - Destroys trap

Matrix and Vector4 Providers

- A “provider” is an abstract interface that is queried for a value of some type
  - Since a provider is a class object, and the query method is virtual, the concrete provider class can be as dynamic as it wishes
  - This paradigm is used by BigWorld to allow continually changing values to be fed from one object to another
    - Objects don’t need to know about concrete provider types
  - Allows Python to setup controllers without requiring any Python code to be ticked per-frame

Matrix and Vector4 Providers

- Vector4 providers
  - Some examples:
    - **`Vector4Animation`** – interpolates between two or more Vector4's over time
    - **`Vector4LFO`** – provides a waveform over time
    - **`Vector4Morph`** – morphs between two values over time
- Matrix providers
  - Some examples:
    - **`MatrixAnimation`** – interpolates between two or more matrices over time
    - **`PyModelNode`** – a matrix provider that represents a joint's transform in world space
    - **`MouseTargetingMatrix`** – provides a world space direction matrix representation of the mouse cursor (i.e. for picking)

Example Vector4Provider Usage

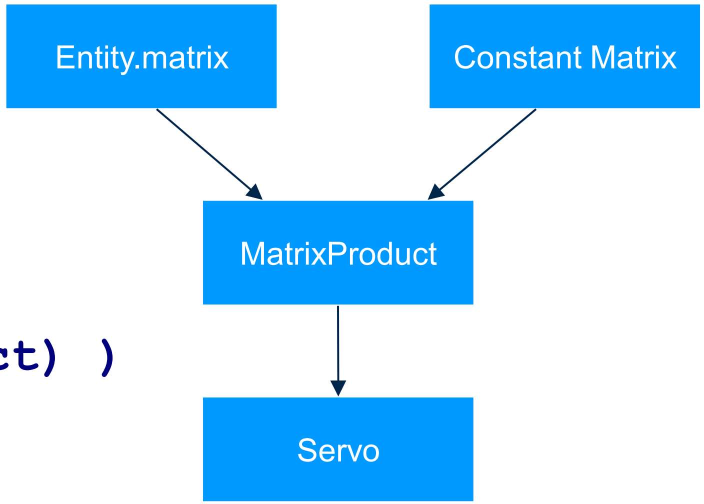
- Animating exposed shader parameters
  - e.g. Vector4LFO to create a pulsating effect on a model:

```
lfo = Math.Vector4LFO()
lfo.period = 0.5
lfo.waveform = ‘SINE’
$p.model.Single_material_skinned = ‘Merchant’
$p.model.Single_material_skinned.clothesColour3 = lfo
```

Example MatrixProvider Usage

- Moving a model using a motor
  - e.g. make a box that floats 2 metres above the player

```
model = BigWorld.Model("sets/town/props/axe.model")
t = Math.Matrix()
t.setTranslate( (0,2,0) )
product = Math.MatrixProduct()
product.a = $p.matrix
product.b = t
model.addMotor( BigWorld.Servo(product) )
$p.addModel( model )
```



Demo Time

- Presented:
  - Direction Cursor
  - Targeting
  - Traps
  - Matrix and Vector Providers
  - Demo App:

Session 8
Particle Systems and Effects

Particle Systems and Effects - Overview

- Particle Systems
- Flora
- Weather
- Water
- Post Processing

Particle Systems

- **SpriteParticleRenderer()**
  - Renders each particle as a single sprite
- **PointSpriteParticleRenderer()**
  - Renders each particle as a pointsprite (maximum size of 64 pixels)
- **AmpParticleRenderer()**
  - Draws a series of wiggly lines back to the source to simulate, for example, electricity
- **BlurParticleRenderer()**
  - Draws particle trails (cheap)
- **TrailParticleRenderer()**
  - Draws particle trails (more expensive)
- **MeshParticleRenderer()**
  - Renders each particle as a mesh object

Particle System Actions

Action	Description
Source	Creates particles over time, on demand or on move
Barrier	Forms an invisible shape to reflect or stop particles
Stream	Converges the velocity of particles to a stream
Scaler	Scales a spawned particle over time
NodeClamp	Links particles to the **`PyModel`** Node to which the
Flare	Draws a lens flare from one or more particles
Splat	Similar to Collide, but calls into script per-collision
Magnet	Accelerates particles towards a particular point

Using Particle Systems

- **`Pixie.create(`*`fileName`*`)`**
  - Returns a ***`ParticleSystem`*** or ***`MetaParticleSystem`***
  - Loads definition from XML file ***`fileName`***
- **`Pixie.ParticleSystem(`*`capacity`*`)`**
  - Creates an empty **`ParticleSystem`**
  - Initial capacity is ***`capacity`*** particles (default is 100)
- **`Pixie.MetaParticleSystem()`**
  - Creates an empty **`MetaParticleSystem`**
- **`model.rightHand = `*`ps`***
  - Attaches **`ParticleSystem`** *ps* to the **`rightHand`** node

Weather

- Low level weather API exposed to Python:
  - **`BigWorld.addSkyBox`** – fade in/out sky box models
  - **`BigWorld.weatherController`** – control rain streaks
  - **`BigWorld.sunlightController`** – sunlight colour multiplier
  - **`BigWorld.ambientController`** – ambient lighting colour multiplier
  - **`BigWorld.fogController`** – fog colour and density multiplier
- High level weather API, implemented in Python
  - XML based weather patterns
  - Fully editable in World Editor
  - Smoothly transition between weather patterns
- Clients are kept in sync via the environment sync
- Create localised weather systems by creating a weather entity

Flora

- A system that allows lots of small detail objects to be rendered efficiently within a radius around the player
  - e.g. grasses, ferns, pebbles, debris
    - The shape of individual flora objects are defined by .visual files.
  - Non-interactive (i.e. no collision between player and flora)
  - Can be animated over time using a Perlin noise generator
- Fades in/out by distance as the player moves
- Placement is automatic, driven by the underlying texture on the terrain
  - e.g. a grass texture would create grass floras
  - Texture -> flora mappings defined in the flora XML file

Water

- Individual water bodies placed in the WorldEditor
- Tweakable shader parameters
  - e.g. colour, fresnel exponent, texture scale, ripple size/speed/direction
- Reflection/refraction
- Normal map based ripple physics simulation that responds to objects moving across the surface
- Uses MRT depth buffer to create soft edge and foam effects
  - High-end feature

Post Processing

- Fully customisable post processing chains
  - A Chain is a list of Effects
  - An Effect is a list of Phases
- Editable in World Editor
- One Chain is active at any one time
  - Effects within a chain can be turned on and off (“bypass”)
  - Order is important to define how effects stack
- A Phase is defined by a pixel shader
  - The output of one Phase is fed into the next

Session 9
GUI

GUI - Overview

- GUI components
  - Provides a hierarchy of 2D quads which render to the screen in correct order, and which detect input events
  - Can be serialised to/from an XML file
- Component scripts
  - A Python class instance that is attached to a particular component in order to handle logic
  - e.g. button, slider, action bar
- Input handling
  - Component posts keyboard and mouse events to the attached script object
- Mouse cursor

Component Types

- **Simple(*texture*)**
  - ***texture*** can be a path to a texture, or a **PyTextureProvider**
  - All other component types derive from this type
- **Window(texture)**
  - Children inherit the position of the Window
  - Clips children to the region of the Window
  - Can be used to scroll the children within the Window
- **Frame2(*texture*)**
  - A resizable frame that avoids texture stretching
- **BoundingBox(*texture*)**
  - Renders ***texture*** at the corners of an entity's bounding box (projected to screen)

bigWORLD TECHNOLOGY

Component Types

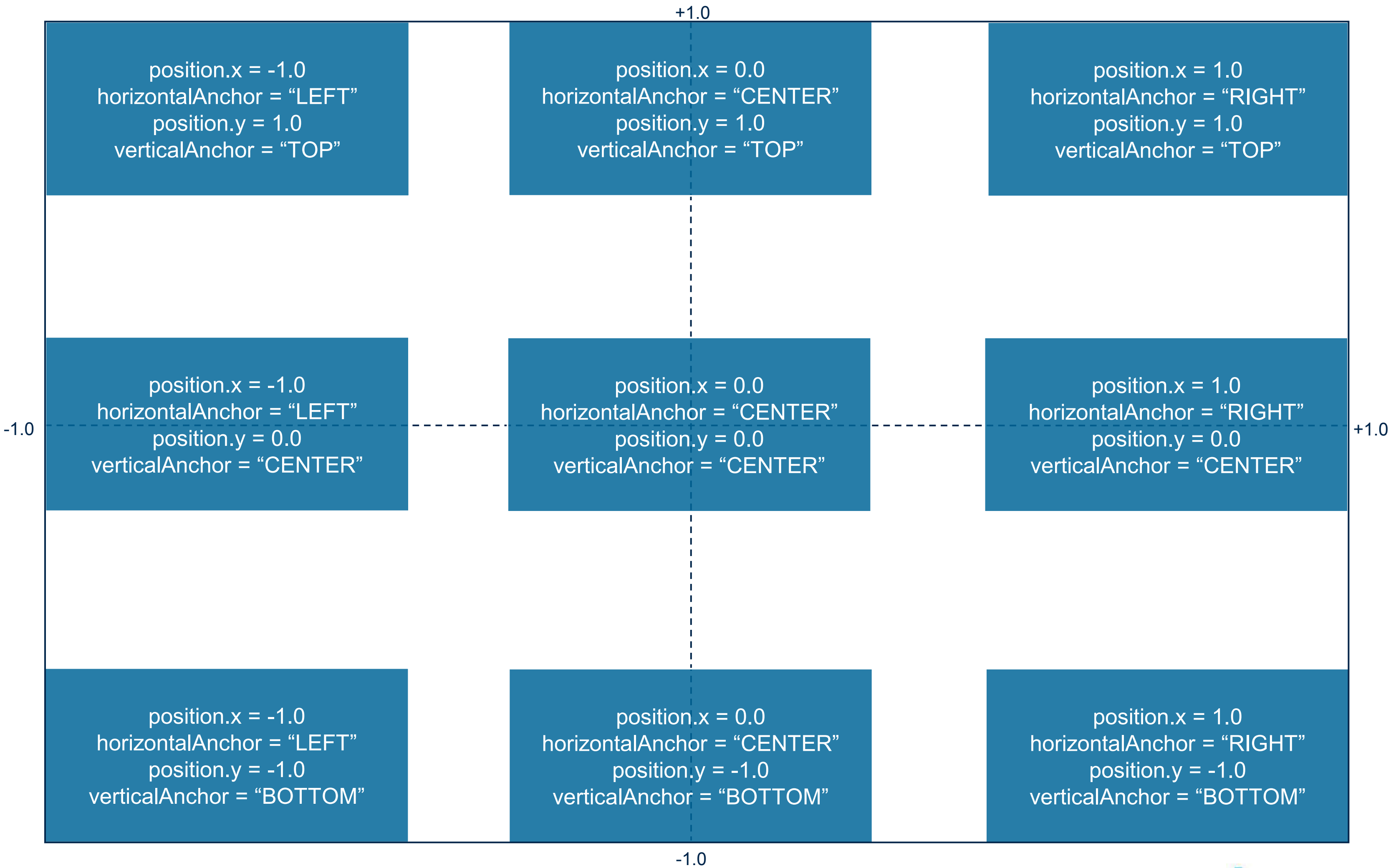
- **`Text(`*`text`*`)`**
  - Renders a line of text using a bitmap font
  - The currently used bitmap font can be changed at runtime
- **`MeshAdaptor()`**
  - Contains a **`PyModelAttachment`** to be included in the GUI tree

Component Layout

- By default, components are defined in clip space (position and size)
  - So 0, 0 is the centre of the screen and +Y is up and +X is to the right
  - The clip space ranges from -1.0 to +1.0 on both axis, so the screen is 2.0 units wide and 2.0 units high
  - Allows a GUI to be designed to be resolution independent
  - Also allows a component position to be anchored at different locations
    - e.g. a component can have its horizontal clip space position set to 1.0, with its horizontal anchor set to RIGHT in order to have a component float on the right hand side of the screen.

Component Layout

Clip space position and anchors



Component Layout

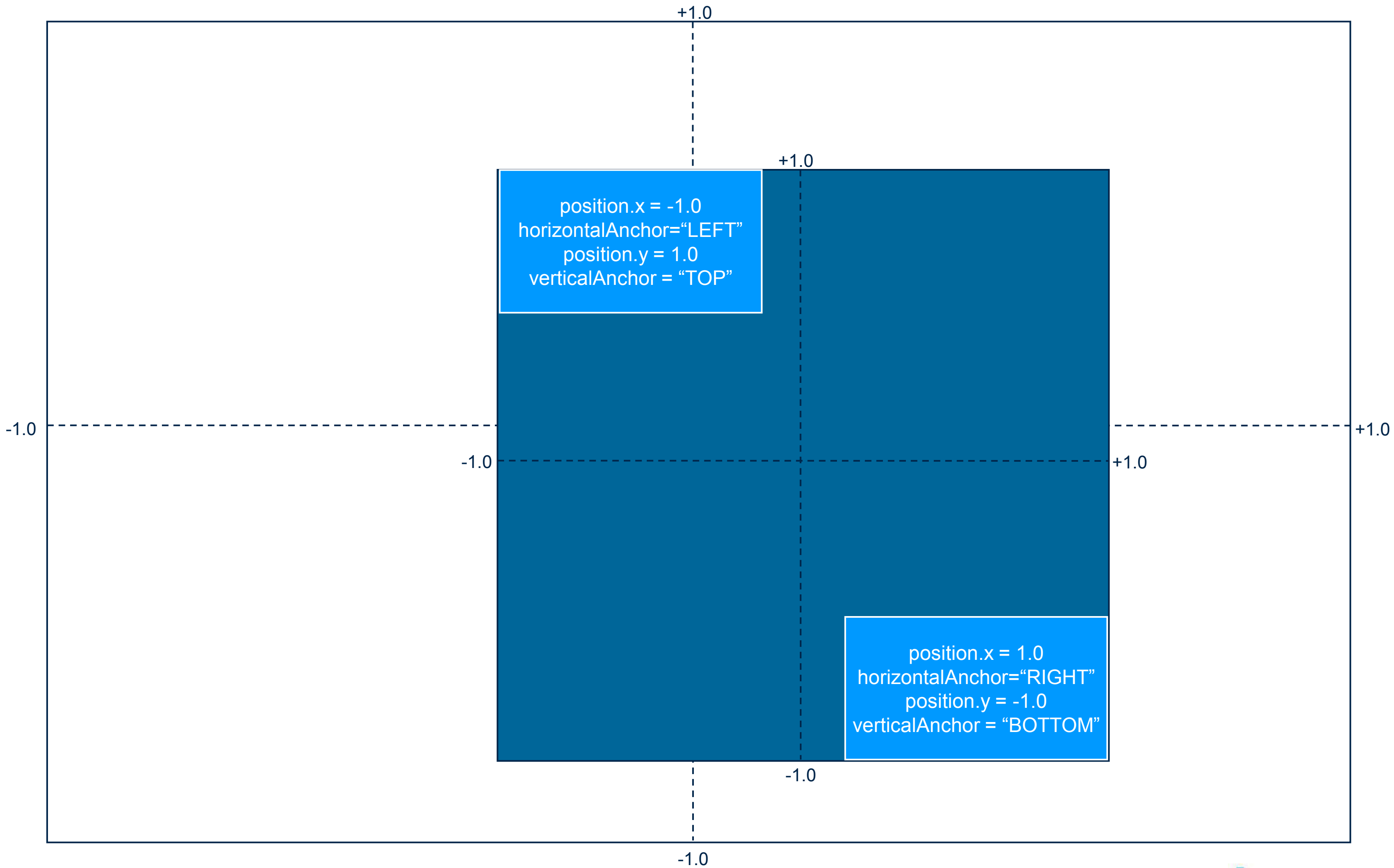
- A Simple GUI component does not inherit position from its parent…
- …unless the parent is a Window component
  - If a child is defined in clip space, then the clip space is contained within the window
    - So 0,0 is the centre of the window, -1.0 is the left side of the window, +1.0 is the right side of the window, etc.

Component Layout

- You can also define a component’s position or size to be in pixels
  - Relative to the top left of the screen (or Window, if it is a child of a Window component)

Component Layout

Window transform inheritance



GUI Shaders

- Modifies the appearance of a GUI component
  - Not to be confused with vertex/pixel shaders
  - Can attach multiple shaders to a single component
- **`AlphaShader()`**
  - Controls the alpha blending of a GUI component
- **`ClipShader()`**
  - Clips a component. Example: a progress bar
- **`ColourShader()`**
  - Dynamically changes or adjusts colour of a GUI component
- **`MatrixShader()`**
  - Moves and resizes GUI components

bigWORLD TECHNOLOGY

Example

- Create and configure the Components

```
guiBG = GUI.Simple("bg.bmp")
guiBG.position = (-1, -1, 0.5)
guiText = GUI.Text("blah")
guiText.position = (-1, -1, 0.4)
guiText.font = "default_small.font"
```

- Add the components to the GUI tree

```
guiBG.addChild(guiText)
GUI.addRoot(guiBG)
```

- Apply shaders to GUI

```
alphaBlend = GUI.AlphaShader()
guiBG.addShader(alphaBlend)
```

Attaching GUI to Models

- **`GuiAttachment`** inherits from **`PyAttachment`**
  - Can be attached to a **`PyModelNode`**
  - A GUI component can then be attached to the **`GuiAttachment`**
- Example

```
guiBG = GUI.Simple("bg.bmp")
guiAttach = GUI.Attachment()
guiAttach.component = guiBG
model.rightHand = guiAttach
```

Input Handling

- A GUI component script can handle input events by setting the appropriate flags on the component and implementing callbacks as required
- Input Callbacks:
  - handleKeyEvent(…)
  - handleMouseEvent(…)
  - handleAxisEvent(…)
  - handleMouseClickEvent(…)
  - etc

Mouse Cursor

- The MouseCursor object encapsulates all cursor behaviour
- Cursor shapes are defined via the mouse cursors definition file…
  - default file: gui/mouse_cursors.xml or
  - specify file in resources.xml under gui/cursorsDefinitions
- …or you can define a mouse cursor shape from a PyTextureProvider
  - for example, you can combine this with a PyModelRenderer to have a dynamically rendered 3D mouse cursor

Demo Time

- Presented:
  - GUI Components
  - Input Handling
  - Mouse Cursor
  - Demo App:

Session 10
Job System

Job System - Overview

- CPU Core Utilisation
- Jobs
- Direct3D Wrapper
- Synchronisation Blocks

CPU Core Utilisation

- Work is split up and offloaded onto multiple cores, if available
  - **Core 1**: Main Thread
  - **Core 2**: Rendering Thread
  - **Core 3**: Background Loading Thread
  - **Core 4..N**: Job Threads
- Dual core is treated as a special case
  - **Core 1**: Main Thread
  - **Core 2**: Rendering and Job Thread
  - Background Loading Thread is allowed to roam
- Single core, job system is disabled

Jobs

- A Job is a discrete unit of processing which is offloaded onto one of the job cores
- Jobs always start in order but can finish in any order
- Each job core will pull the next available job from the queue as their previous job finishes
- Each job can be given a pointer into which to write output data which will become valid
    - Created using **`JobSystem::allocOutput`**

Direct3D Wrapper

- Direct3D runs on its own core
  - This helps prevent the video card from being starved of work
- The device is wrapped at the API level, which is transparent to the user
  - e.g. can call **`DrawPrimitive`** as per usual without worrying about when and where it will actually be called on the device
  - Calls are queued up onto a command buffer for execution on the next frame

Direct3D Wrapper – Resource Locking

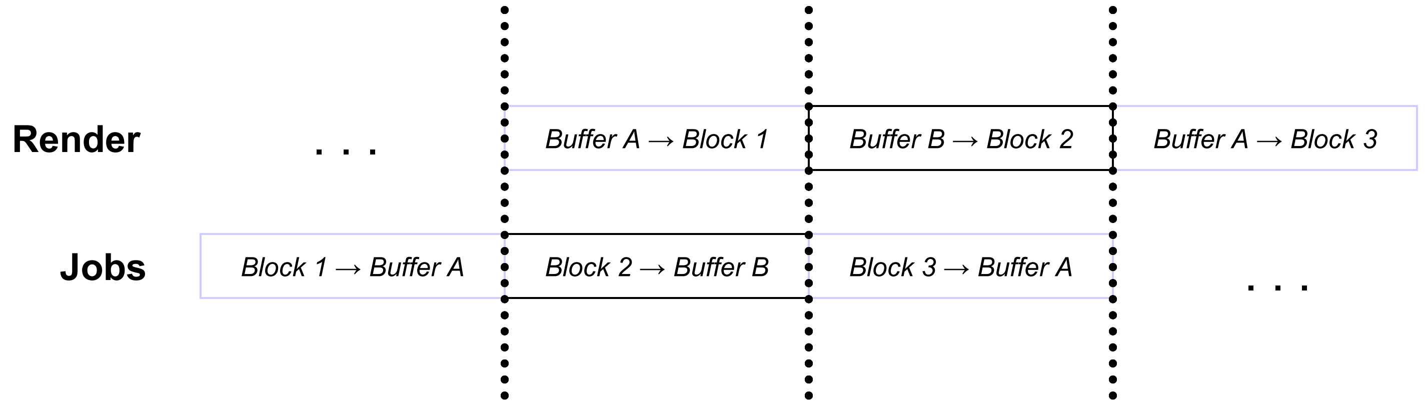
- Resources such as vertex buffers can make use of deferred locking
  - This is used for filling buffers in one or more Jobs.
  - Allocated by **`JobSystem::allocOutput`**, so the lock pointer must only be used later in a job's execute method!

Synchronisation Blocks

- Each frame is divided into “blocks”
  - Rendered in order, one after the other (synchronously)
  - Each block has a set of associated D3D commands and jobs
  - Any number of jobs can be setup to produce output for a particular block

Synchronisation Blocks

- Jobs are one block ahead of the rendering thread
  - So hopefully the jobs are complete by the time the rendering thread catches up!
  - If the rendering thread gets to the block barrier first, it will wait until all jobs are complete
- Job output is double buffered, hence they can only ever be one block ahead



Synchronisation Blocks

- Typical usage:
  - Create new block
  - Lock vertex buffers using deferred locks
  - Create N jobs
  - Setup render states
  - Issues draw calls
- Note that order doesn't matter: the jobs will always finish execution before the rendering occurs

Session 11
Profiling and Engine Statistics

Profiling and Engine Statistics - Overview

- Real-time performance monitoring
- Deterministic profiler tests
- Soak testing
- Watchers

Real-time Performance Monitoring

- Real-time Profiling Console
  - Accessed in-game via DEBUG+F5
  - Displays various statistics, averaged over the last few frames
    - Frame rate
    - Primitive counts
    - Breakdown of Dog Watcher hierarchy
- Dog Watchers
  - Named sections of code marked for timing
  - Allows for hierarchical profiling of different parts of the code

Profiler

- Camera follows a specific path for a fixed number of frames
- Outputs per-frame details into a CSV spreadsheet for analysis
- Can run in “Profiler History” mode
  - Similar to standard profiler, but the game remains interactive
  - Useful for testing specific cases that cannot be automated

Soak Testing

- Soak testing
  - Runs a camera fly-through for a certain amount of time
  - Frame rate is not limited – provides average frame rate at the end
  - Used to test whether the game will run for long periods of time
  - Outputs MemTracker stats every 6 seconds
  - Outputs CSV file as in the profiler test, but includes memory totals

Watchers

- Watchers are internal engine variables that can be monitored via the watcher screen (DEBUG+F7)
  - e.g. network subsystem statistics are exposed as watchers
  - Stored by category, hierarchically
  - Use PGUP and PGDN to scroll between watcher values
- Watchers can be Read Only or Read/Write
  - Changing certain watcher values can tweak engine behaviour
  - Provides access to some debug functionality
    - e.g. visualisation of skeletons, portals, etc.
- Accessible from Python via `BigWorld.setWatcher` and `BigWorld.getWatcher`
- Not available in Consumer Release build

Conclusion

- This completes the Client training
- Detailed information can be found in the Client Programming Guide and the Client Python API documentation
- Thanks for attending

bigWORLD TECHNOLOGY