BENEATH THE BADLANDS

Technical Design Document

DEADBOLT
INTERACTIVE

Table of Contents

(Click anywhere on a line to skip to its section.)

Software Versions

Whenever possible, software versions on home computers should match those used on school computers. Never update assets to work with software versions which do not work on the computers in Montgomery hall!

Software	Version
Visual Studio	2022
Unreal Engine	5.1.x
Substance Painter	8.2.0
Substance Designer	12.3.0
Autodesk Maya 2020	2020.4
Photoshop	24.0
Reaper	Info Pending
Blender	3.2
Zbrush	2022.0.5

Naming Conventions & Folder Structure

*Most of this information is taken from UE's Recommended Asset Naming Conventions with some small modifications.**

This document is subject to change until production starts on assets, unless there is a major problem or there was an overlooked asset type.

Naming Conventions

Most file names will follow this name structure:
`[Type Prefix]_[Asset Name]_[Descriptor/Variant]`

Asset names should be descriptive and concise!

Asset Types

Click a button to skip to a specific type page.
Use Ctrl + F to search for a specific term.

Materials
Textures
Meshes
Particles
Blueprints
Misc.

Meshes

Asset Type	Naming Convention
Static Mesh	SM_[Asset Name]
Skeletal Mesh	SKM_[Asset Name]
Physics Asset	PHYS_[Asset Name]
Control Rig	CR_[Asset Name]
Animation, Descriptor	ANIM_[Asset Name]_[Descriptor]

Materials

Asset Type	Naming Convention
Master Material	M_[Generic Name]
Material Instance	MI_[Specific Name or Asset Name]
Post Process Material	PPM_[Name]
Material Function	MF_[Function Name]

Particles

Asset Type	Naming Convention
Niagara Emitter	NE_
Niagara System	NS_
Niagara Function	NF_

Textures

`T_[Asset Name]_[Descriptor/Variant]_[Type Suffix]`

Asset Type	Suffix
Diffuse, Alpha	_D
Normal	_N
Occlusion, Roughness, Metallic Packed	_ORM
Emissive	_E
Packed Masks	_M
UI Elements	_UI

*Make sure to only export the required textures.**

Packed Masks

Red Channel:
Displacement

Green Channel:
Subsurface Scattering Mask

Blue Channel:
Specular

Blueprints

Asset Type	Naming Convention
Blueprint	**BP_**
Animation Blueprint	**ABP_[Asset Name]**
Blueprint Interface	**BPI_**
Widget Blueprint	**WBP_**
Structure	**ST_**
Enumerator	**E_**
Curve Data Asset	**C_**
Behavior Tree	**BT_**
Blackboard	**BB_**
Blueprint Function Library	**BFL_**
Blueprint Macro Library	**BML_**

Misc.

Asset Type	Naming Convention
Sequence	**SEQ_**
Level	**L_**

Handling Duplicate Names

This is what the Descriptor/Variant part of the file name is for. If you are making something like leaves and you make multiple kinds, give them a variant name or number.

Ex.
`T_Leaf_Oak_D`
`T_Leaf_1_D`
`T_Leaf_Oak1_D`

Duplicate Meshes

There should be no duplicate meshes, instead create separate actors that use the same mesh and assign the materials in the actor. These are placed in the `/Blueprints/Actors` folder.

Modules

In lieu of a typical “geometry, texture, material, etc.” folder structure under Content, assets are divided into modules. Modules are designed as largely freestanding sections of the game. The purpose of modules is to help avoid bloated asset folders in our large team project, and reduce the number of people accessing one folder at a given time. Modules are stored in the Content folder, or as plugins. While they will not adhere to our naming conventions or folder structure, marketplace content and externally developed plugins are still considered Modules.

A module might be a level, a collection of enemy blueprints, a collection of weapon blueprints.

Modules may depend upon other modules or have modules who depend on them.

A module is “checked out” by a scrum team, and can remain checked out for as long as they desire. A module “checked out” by a scrum team cannot be edited by another team without their permission.

Modules, and information on them are stored in the Module Spreadsheet, which lists all modules, their dependencies, and which team has them checked out.

Continued on next page.

Modules Cont.

Modules should strive to be as freestanding as possible without inhibiting workflow. In an ideal world, a module and all its assets should work without any other modules, except those explicitly stated as dependencies. However, duplicate assets should also be avoided. In those cases where an asset from another module is needed, consider moving the asset to the _SharedAssets module with the permission of the module's current owner or listing the module containing the asset as a dependency.

The _Core module contains base blueprint classes needed throughout the project, as well as the player character, and base materials.

The _SharedAssets module contains assets expected to be used in multiple modules. The _SharedAssets module is an exception to module checkout rules and can be added to by any scrum team at any time.

Other modules include:

- AI
- Weapons
- Biome 1
- Biome 2
- Biome 3
- Badlands
- Procedural
- Mining Town

Finding assets among modules (or the large folders created by a module-less system) can be hard. Thankfully, Unreal Engine's search and filter tools are quick and powerful. Teach yourself to find assets by default by searching for them rather than trying to navigate through folders. Start by filtering for the asset type (or start your search with the appropriate prefix), then search likely terms if you don't know the asset name.

Folder Structure In Module

Each module contains:

Base Folders

```
ModuleName
├──Assets
├──Blueprints
├──Cinematics
├──Maps
└──Sounds
```

Assets

All visual assets will be placed into an ‘Assets’ folder with sub-folders to separate by type and asset.

```
Assets
├──Characters
├──Environments
├──Foliage
├──Textures
├──UI
```

Folder Structure In Module Cont.

Characters

Holds all character assets separated into folders by type and subfolders for each character.

```
Characters
├──Player
│   └──Fox
│       ├──Animations*
│       │       ANIM_Fox_Jump
│       ├──Effects
│       ├──Materials
│       │       M_Fox
│       ├──Meshes
│       │       ABP_Fox
│       │       PHYS_Fox
│       │       SKEL_Fox
│       │       SK_Fox
│       └──Textures
│               T_Fox_D
│               T_Fox_M
│               T_Fox_N
│               T_Fox_ORM
├──Enemy
└──NPC
```

Note*
Animation Blueprints go with the corresponding mesh

Environments

Holds environment assets separated by environment with subfolders for Props and Terrain. Global holds universal environment assets.

```
Environments
├──Town
│   ├──Effects
│   │   ├──Emitters
│   │   ├──Functions
│   │   ├──Materials
│   │   ├──Meshes
│   │   ├──Textures
│   │   └──Systems
│   ├──Props
│   │   ├──Materials
│   │   ├──Meshes
│   │   └──Textures
│   ├──Terrains
│   │   ├──Materials
│   │   ├──Meshes
│   │   └──Textures
│   │   ├──Meshes
│   │   └──Textures
│   └──Foliage
├──Swamp
└──Global
```

Folder Structure In Module Cont.

Effects

Holds all effects for the area.

Props

Holds anything that isn't repeated and/or is complex.

*Terrain**

Holds anything that is repeated and modular pieces.

*Foliage**

Holds all foliage assets separated into materials, meshes, and textures folders.

```
Foliage
├──Materials
│       MI_Leaf
│       M_Foliage
├──Meshes
│       SM_Leaf
└──Textures
        T_Leaf_D
        T_Leaf_M
        T_Leaf_N
        T_Leaf_ORM
```

Note*
Foliage and Terrain do not separate each asset based on name

Textures

Holds all miscellaneous textures that are used across the project, ie default normals, textures, alphas, noise.

UI

Holds all UI visual assets. Does not hold widget blueprints.

```
UI
├──Materials
└──Textures
```

Blueprints

Holds all blueprints (excluding animation blueprints). Blueprints are separated by purpose.

```
Blueprints
├── Category 1
├── Category 2
├── etc.
```

Cinematics

Holds all sequences

Maps

All maps for the module

Sounds

All sounds for the module

```
Sounds
├── SFX
├── Music
```

Asset Storage

Assets will be stored in Perforce under the AssetStorage folder. Within the AssetStorage folder will be a folder for each module, and the folder structure within that module will replicate that of the module, with some changes.

Base Folders

```
ModuleName
├──Assets
└──Sounds
```

Assets

All visual assets will be placed into an ‘Assets’ folder with sub-folders to separate by type and asset.

```
Assets
├──Characters
├──Environments
├──Textures
├──UI
```

Characters

Holds all character assets separated into folders by type and subfolders for each character.

```
Characters
├──Player
│   └──Fox
│       ├──Animations
│       │       ANIM_Fox_Jump.fbx
│       ├──Meshes
│       │       Fox.ztl
│       │       Fox.ma
│       │       Fox.mb
│       ├──Substance
│       │       Fox.spp
│       │       Fox.sbs
│       │
│       └──Textures
│               T_Fox_D
│               T_Fox_M
│               T_Fox_N
│               T_Fox_ORM
├──Enemy
└──NPC
```

Textures

Holds all miscellaneous textures that are used across the project, ie default normals, textures, alphas, noise.

Asset Storage Cont.

UI

Holds all UI visual assets.

```
└──Textures
```

Environments

Holds environment assets separated by environment with subfolders for Props and Terrain.

```
Environments
├──Town
│   ├──Effects
│   │   ├──Meshes
│   │   └──Textures
│   └──Props
│       ├──Meshes
│       │       Bench.ma
│       │       Bench.fbx
│       └──Substance
│               Bench.spp
│               Library.sbs
│
├──Swamp
└──Global
```

Collections

https://docs.unrealengine.com/5.1/en-US/filters-and-collections-in-unreal-engine/

Collections are like pallets of assets which can be used to quickly access resources important to a task. A single asset can be in multiple collections! We will have at minimum a collection for each level, as well as one for effects assets and one for UI assets.

All assets added to the project should belong to at least one collection, and assets should be added to appropriate collections at import time.

Note that filtering by asset type will make using collections as pallets much easier!

Level Pipeline Guide

Link To Miro Board

Approval Processes

During approval steps, all listed persons are given the chance to accept or reject the work, after discussing it with the person who performed the work, and the other approvers. If work is accepted, proceed to the next pipeline step. If it is rejected, the approvers identify how far the character needs to move backwards in the pipeline for revisions.

Level Concept

1) **Level Idea**
   There is a proposed concept that needs supporting visuals before it can be implemented in the game. Teams must brainstorm and agree on a concept to be developed.
2) **Moodboard**
   A moodboard is created to establish the general atmosphere desired for a level and to establish visual references to be used during level conception and creation.
3) **Thumbnails**
   Based on the created moodboard, thumbnails with value comps should be sketched proposing a variety of different compositions involving changes in lighting, prop placement, architecture, particles, etc.
   a) **Thumbnail Approval**
      Mae will approve a thumbnail to be developed further.

Level Concept Cont.

4) **Multiple Camera Angle Sketches**
   Based on the approved thumbnail, multiple compositions depicting various camera angles of the environment are developed.
   a) **Multiple Camera Angle Sketches Approval**
      Mae will approve a composition that best captures the level's desired design. The compositions that are not chosen will still be delivered to the environment team to serve as supporting alternative views of the environment.
5) **Final Render**
   The approved composition is rendered in color and the lighting and details are polished.
   a) **Final Render Approval**
      Mae, with Nat and Jacob's input, as well as Coral and Matt, will approve the composition to be delivered to the environment team alongside the alternative camera angles from step 4.
6) **Particle Mood Board**
   A moodboard is created to establish visual references to be used during particle conception and creation.
   a) **Particle Mood Board Approval**
      Mae, with Kano's input, will approve the moodboard to be delivered to the particle team.

Level Creation

1) **LDDs (*Town/Badlands, Mines, Nests, Blood Pits*)**
   Overview of all aspects pertaining to the Level.
   Each LDD features level specific hazards, interactables, and obstacles. However all LDDs share similar room types. These room types will be closely worked on throughout the Top-Down phase (2) as each will be vital for capturing the designed level's flow, theme, and experience. The LDDs will be the key reference for the designers to use as it explains the specific levels goals and design specifics, narratively and gameplay wise.

   a) **Approval (by Phil/Eric)**
   Phil and Eric will review and approve each level's Design Document along with input from the document's creator.

2) **Level Design Drawings (*Top Down*)**
   Regarding the LDDs; drawings will be of all individual level room types: Close Quarters, Hybrid, Long Distance, Arena, and Treasure Rooms along with focus on the specific flow per level we want experienced. Each top down will follow a specific detailed illustrated kit that is in each LDDs Environment section in order to correctly and cohesively translate the design into the Rapid Prototyping phase (3) and the Blockout phase (4) . All finalized sketches will be refined in either illustrator or miro and then added to a folder in dropbox.

   Specified naming conventions for the organization of these top downs are the following:
   XX represents the files number ie. 01, 02, 34)
   LevelName_Roomtype_XX.png

   a) **Approval (by Phil/Eric)**
   Phil and Eric will review and approve each level's top-down collection along with input from the illustrator's creator.

Level Creation Cont.

3) Rapid Prototyping (*Primitives*)

For scale, proportion, player movement, and angles, the rapid prototyping is the first step of the IN ENGINE build. Using the primitive shapes along with the UE5 Brushes the levels will be primitively blocked in order to capture a closer resemblance to each top down drawing. Each level's room will be its own separate level, later used in the main levels map through the procedural build system.

The following naming convention is used for a specific level's rooms:
LevelAbbreviation_RoomType_LVL

The level abbreviations for each level are as followed:
MNS = The Mines
NST = The Nests
BLP = Blood Pits

The following naming convention is used for a specific level in its entirety:
LVL_LevelName

a) Playtest Loop (by a Level Designer)

Initial test for space, flow, and angles. This will force continuous iterations on the level and allow for more practical fixes prior to a final blockout. Using perforce all changes MUST be documented in the changelogs. Screen captures are heavily recommended in order for process documentation and changelog history when it comes to changes, however they are MANDATORY for actual level tests.

Any screen recordings whether it is a test or changelog will share the following naming conventions:
RP_LevelAbbreviation_Test_Roomtype.mp4
RP_LevelAbbreviation_Change_Roomtype_.mp4

All recordings will be placed into a designated folder inside of dropbox or google drive inside of a folder labeled *Rapid Prototyping* then into a second folder under either *Test* or *Change*

Level Creation Cont.

4) Blockout

a) Playtest Loop (by a Level Designer)

Base testing loop for all things pertaining to gameplay; enemies, angles, spacing, movement, interactions, hazards, and completion. Everything, besides art, will be placed and tested thoroughly before going to the next phase. Using perforce all changes MUST be documented in the changelogs. Screen captures are heavily recommended in order for process documentation and changelog history when it comes to changes, however they are MANDATORY for actual level tests.

Any screen recordings whether it is a test or changelog will share the following naming conventions:
BO_LevelAbbreviation_Test_Roomtype.mp4
BO_LevelAbbreviation_Change_Roomtype_.mp4

All recordings will be placed into a designated folder inside of dropbox or google drive inside of a folder labeled *Blockout* then into a second folder under either *Test* or *Change*

b) Approval (by Phil/Eric + Matt & Coral)

Phil and Eric will review and approve each level's Blockout along with input from the designs creator.

5) Design Blueprints for Level

This is the base design for all things regarding the levels interactions as well as completion. Hazards, obstacles, sequences, and spawns. The design will be documented prior to creation in engine. The documentation will be inside of each LDD under the *Level Blueprints* section.

Level Creation Cont.

6) Create Blueprints

This is the creation of the designed BPs regarding the level. Testing is beneficial during creation however, most in level testing will be done during the (4) Blockout phase.

Level BPs in regards to the specific levels LDD will be broken into 3 categories. Hazard, Intractable, obstacle. The naming conventions are as followed:
LevelAbbreviation_BPType_BP
Any level BP that is not in the 3 above categories will be named:
LevelAbbreviation_RoomType_BP

a) Blueprint Approval (by Phil/Eric)

Phil and Eric will review and approve each level blueprint along with input from the blueprint's creator.

Asset List Development

1) **Create Particle List**
   A list will be compiled of all needed particle effects to satisfy the narrative and visual needs of the level. These should be based on the needs of interactivity within the level.
2) **Sound Particle List**
   a) **Particle List Approval**
3) **Create Prop List**
   Referencing the concept art, a list will be made of all individual props needed for the level
4) **Prop Sound List**
5) Prop Concept Art (*for narrative-driving props*)
   a) **Prop List Approval**
6) **Create Module List**
   Referencing the concept art, a list will be made of all modular assets needed for the level
7) **Ambient Sounds List**
8) **Module Concept Sketches**
   a) **Concept Sketch Approval (w/ Matt & Coral)**
9) **Create Material List**
   All the needed materials for the level will be considered based on real life factors and player needs.
   a) **Material List Approval**
10) **Create Material Assets**
    Material Assets, including decals and terrain, will be created in designer and imported into unreal.
    a) **Material Asset Approval**
11) **Implement Material Assets**
    Material assets will be placed, painted, and otherwise placed within the level.

Asset Creation

1) **Create Particle Assets**
   Particle assets being materials, textures, and necessary meshes for Niagara Systems.
2) **Niagara**
   Incorporate particle assets into Niagara Systems.
   a) **Particle Asset Approval**
3) **Particle SFX**
4) **Prop Modeling**
   Referencing the concept art, the environment team will choose props from the created list to model for the level. No UV's yet.
   a) **Prop Model Approval**
5) **Prop Texturing**
   Props should be textured and incorporated into materials within the engine. Approvals will not happen within texturing programs, as materials can look different in-engine.
   a) **Prop Texture Approval**
6) **Module Whitebox**
   The basic shapes of modular assets should be assembled to scale with the same pivot the final version will use.
   a) **Module Whitebox Approval**
7) **Module Modeling**
   Referencing the concept art, the environment team will choose modular assets from the created list to model for the level. No UV's yet.
   a) **Module Model Approval**
8) **Module Texturing**
   UV's will be created, the modular asset will be textured and placed within the engine for review.
   a) **Module Texture Approval**

Sound Creation

1) **Environment SFX List**
    a) **Off to Sound Pipeline**
2) **List of Needed Tracks**
    a) **Off to Sound Pipeline**

Bring in Art, Iterate

1) **Lighting Block**
    Serious pass for lighting. Pretend its final. No builds will be required with Lumen, however time and consideration should go into player pathing and wide cutscene shots.
2) **Reimport Whitebox Modules with Final Art**
    All final assets that have been altered during iterations will be imported and fully integrated in the engine.
3) **Lighting Refinement**
    After textures are in the engine, lighting needs may change slightly. This lighting pass ensures all changes are accounted for.
4) **Texture Iteration**
    Lighting may change texture needs, and textures may not flow together at this stage. This pass is intended to fix those problems.
5) **Set Dressing**
    Fleshing out the scene using premade assets to make the world more believable.
    a) **Art and Lighting Iteration Passes**
        Playtesting and set dressing will alter the needs for lighting. An iterative pass will happen as final adjustments are being made to the level.
    b) **Playtest**
        Level Designers will test and iterate the final collective of art and design, along with gaining input from the artists.
6) **Add Final Art to Blueprints**
    Any art associated with BP assets will be imported and implemented.
7) **Sound Pipeline**
8) **Final Sounds**

Final

1) **Final Level Approval**
    a) **Final Level Approval is performed by the Product Owners (Coral & Matt), design leads (Eric and Phil), and environment leads (Jacob and Nat). Gameplay, lighting, art assets, and all other aspects of the level should be considered.**

2) **Finished Level**

Character Pipeline Guide

[Link To Miro Board]

Approval Processes

During approval steps, all listed persons are given the chance to accept or reject the work, after discussing it with the person who performed the work, and the other approvers. If work is accepted, proceed to the next pipeline step. If it is rejected, the approvers identify how far the character needs to move backwards in the pipeline for revisions.

Character Concept

1. **Idea for Character**
2. **Silhouette Work**
3. **Develop detailed proposals**
4. **Develop multiple variations**
5. **Select and approve final design**
6. **Model sheet**
7. **Approval of final model sheet**

Character Sculpt

7) **Blockout**

The blockout stage must have all primary and secondary forms complete. The general form and proportions of the character must be visible at this stage. The character should also be placed in proper scale and have organized named subtools. The file should also have proper naming conventions. Artists should also be saving multiple iterations and backups.

File Naming Convention:
BTB_CHAR_CharacterName_Blockout_LastnameFirstname_Iteration
Example: BTB_CHAR_Aster_Blockout_ArtersJohn_9.ztl

   a) **Approval (by Shannon/John)**

8) **Final Sculpt**

At this stage, all sculpting must be complete. The sculpt must closely match the concept art and have hair cards (if there are any) placed. All surface detail must be done, and there must be multiple subdivisions. All subtools must be named and organized into folders. The model must be ready for topology. Must export as obj.

File Naming Convention for ZTool: BTB_CHAR_CharacternameFinal_Sculpt_LastnameFirstname
Example: BTB_CHAR_Sage_FinalSculpt_McConnellShannon.ztl

File Naming Convention for Export: BTB_Character name_Lastname
Example: BTB_Sage_McConnell.obj

   a) **Approval (by Shannon/John + Matt & Coral)**

Character Topology & UV

9) Topology

At this point the model should be retopologized to the proper poly count limit (60-80K for hero characters, 30K for everything else) though they should be as low as possible while keeping the shape. The model should also have good edge flow and have edge loops where it will deform. The topology must also be clean with no non-manifold geometry and no Ngons. Absolute limits are 150K for hero characters and 50K for non-hero. This file will be saved in Maya Binary and have backups and iterations.

Topology Guides

General	Hands Hands 2	Feet	Faces	Biped Biped 2	Q-Peds Q-Peds 2	Extra Tools

Naming Convention:
BTB_CHAR_CharacterName_Topology_LastnameFirstname_Iteration
Example: BTB_CHAR_Aster_Topology_ArtersJohn_4.mb

a) Approval (by Shannon/John)

Character Topology & UV Cont.

9) Topology Cont.

In general for animation, it is faster to model a low poly character with clean topology first before sculpting to higher subdivisions, than to do retopology afterwards with a messy high-poly model. That means box modeling, extruding, and keeping an understanding of basic shapes. See the references for examples.

Given only a high poly model, the same rules apply in reverse. Everything should be broken down into basic shapes, and in general everything is either a Cylinder, or a cube Sphere, to maintain all quad topology.

Legs, arms, torso, fingers, are all cylinders, a human type head is spherical. The main problem areas for topology are the intersections of cylinders. The circular edge resolution of a cylinder should be in multiples of two or four. Depending on the details needed, a round cylinder will have anywhere from 8 to 64 edges lengthwise.

As we are using a game engine, we have to use more edge loops for any sharp edge details. Things like the tips of claws, teeth, tentacles, etc, that end in a sharp point, should end in a quad cap, not a single vertex, depending on the size and shape of the detail.

Try to imagine where the joints will bend when placing edge loops, circular edge loops should be as close to flat as possible along their normal axis, no wavy circles. Add more edge loops at bend locations, elbows, knees, finger joints, neck, mouth, hips, etc, and be consistent with the number of loops added, between 2 - 8 extra loops.

Once the basic shape retopology is in place, adjust edges to fit the model details as close as possible while keeping quads regularly spaced.

Draw a band of 8 quad strips around the start of a cylinder, like a Shoulder, extrude all the way to the Wrist, then add more edge loops, as few as possible to maintain the shape. For faces, draw the edge loops around key areas first before connecting the rest, eyes, lips, nose, horns, teeth, tongue, ears, etc, all of these are specific features with their own shape and deformation. For curved shapes, try to keep the edge loops aligned with the direction of the curve, not stretched.

Use the smoothing tool as much as possible without losing details in key areas, if your edge flow is not clean or edges are too dense, you need to smooth it, or remove edge loops. This is harder with more dense topology, so keep it simple.

Character Topology & UV Cont.

10) UV

The UVs must be done on the low-poly model. They can be done in either Maya or Unreal, but they must be clean and make sense. They also must be placed in such a way on the UV tile that makes sense relative to the resolution they'll need. For hero characters, aim for 2 texture sets and for non-hero characters aim for 1 texture set. Absolute limits are 4 texture sets for hero and 2 texture sets for non-hero. Also at this point there should be a Maya file set up for the low-poly model with all of the parts properly named for baking. Must export as fbx.

Naming Convention for Maya File:
BTB_CHAR_CharacterName_LowPoly_LastnameFirstname
Example: BTB_CHAR_Sage_LowPoly_McConnellShannon

Naming Convention for Export: SM_Character name
Example: SM_Sage.fbx

a) Approval (by Shannon/John)

Character Texturing

11) Baking

The high poly model must bake onto the low poly model in Substance Painter with no visible artifacts. Everything must be baked at 4K. The Substance Painter file must also be set up in DirectX and be prepared to export into Unreal with packed textures. The file will also need to have backups and iterations.

Naming Convention for Substance File: BTB_CHAR_Character nameTexture_LastnameFirstname_Iteration
Example: BTB_CHAR_Aster_Texture_ArtersJohn_5.spp

a) **Approval (by Shannon/John)**

12) Texturing

Textures are done to the level of detail desired and match the style demonstrated in the art bible. They must also match the given color pallets. All layers in Substance Painter must be named and organized in folders. Textures must be put in Unreal to get the best idea of what they will look like in game before approval. Textures must be exported into a folder. Export textures as targa files.

There should be three textures per material slot:
BaseColor
Normal
Packed
With packed containing:
G Channel: Roughness
B Channel: Metallic

File folder name: BTB_CHAR_Character name_Textures
Example: BTB_CHAR_Sage_BaseColor

File naming convention:
BTB_CHAR_CharacterName_MaterialSlotName_TextureType
Example: BTB_CHAR_Sage_BaseColor.tga

a) **Approval (by Shannon/John + Matt & Coral)**
b) **Materials must be set up in Unreal to the standards set by the technical director.**

Rigging & Animation

13) Joint Setup

Joints are setup in Maya, complexity will depend on character/creature type. Unreal does have a joint count limit for efficiency, should be no more than 256 joints. Most characters will have a Root Joint that parents all other joints, this is for Unreal setup.

Chains do not have to all be connected, Control Rig works similar to Maya rigging. Limit chain length for things like tails, tentacles, etc, no more than 16 joints in a single chain, target no more than 8.

Re-Use joint setups where possible. Sub-joints can still be used in Control Rig. Clothing/attachment setups may vary. If using physics cloth, only a root joint is needed.

Naming conventions:
C_object_joint_#_offset
L_object_joint_1_CTRL
R_object_joint_2_offset
R_object_joint_end_JNT
R_object_joint_#_bind
C_object_root_CTRL
object_JNT_GRP
object_GEO_GRP

Save with the following naming convention:
Naming Convention:
BTB_CHAR_CharacterName_joints_LastnameFirstname_Iteration
Example: BTB_CHAR_Aster_joints_BrowningJacob_4.mb

Rigging & Animation Cont.

14) Weight Painting

Using ngSkinTools for Maya is recommended. Joints should deform topology smoothly and regularly with no rogue vertices moving on other areas of the model. Work on as few joints at a time as possible with all others locked for default Maya painting, work in the correct Layers with ngSkinTools. Two joints should deform to a 'heart' shaped crease as much as possible, long joint chains may behave differently for tails, tentacles, etc. Topology should have clean curves when posed, no jagged lines.

Save with the following naming convention:
Naming Convention:
BTB_CHAR_CharacterName_Weight_LastnameFirstname_Iteration
Example: BTB_CHAR_Aster_Weight_BrowningJacob_4.mb

Painting Weights Guide

15) Import to Unreal

Import from Maya into Unreal as an .fbx by using export or Game/Unreal export. Check Export UVs, vertex colors.

When exporting, be sure to have everything named appropriately, and have everything selected.

Save with the following naming convention, there should only be one Export file:
Naming Convention:
BTB_CHAR_CharacterName_export_LastnameFirstname.fbx
Example: BTB_CHAR_Aster_export_BrowningJacob.fbx

Import into the game project into the appropriate folder as per the naming convention guides.

Rigging & Animation Cont.

16) Control Rig

Skeletal meshes can have multiple control rigs for different uses. Control Rig can be additive to existing animation. Base rig setup should be similar to Maya so that animators are familiar with the same setup.

One major difference is that Unreal has Space Switching built into the animation editor by default, so for some things there is no need to create a rig space switch, this can be handled by the Animator as needed. Attached items like hats, weapons, etc, can be parented outside the main rig hierarchy, and have a default Space and any number of additional Spaces, world, joint, offset group, etc.

Backwards solve does not need to be setup for the base rig.

Make use of the Unreal solvers, Spine IK, 2-Bone and 3-Bone IK for legs, CCDIK for long joint chains, FABRIK, FBIK, etc.

We will not be using an FK/IK switch setup unless it is needed for animation purposes. A simple FK rig can be enabled for any Skeletal mesh by adding it in Sequencer

Unreal has a lot of Control shapes by default, but more can be added with a Control Shape Library, which can add any mesh as a control shape.

Prefix: CR_assetName

16) Control Rig Cont.

Unreal 5 Documentation:

Control Rig
https://docs.unrealengine.com/5.0/en-US/rigging-with-control-rig-in-unreal-engine/

Space Switching
https://docs.unrealengine.com/5.0/en-US/re-parent-control-rig-controls-in-real-time-in-unreal-engine/

Spline rig
https://docs.unrealengine.com/5.0/en-US/control-rig-spline-rigging-in-unreal-engine/

Custom Control Shapes
https://docs.unrealengine.com/5.0/en-US/control-shapes-and-control-shape-library-in-unreal-engine/

FK Rig:
https://docs.unrealengine.com/5.0/en-US/fk-control-rig-in-unreal-engine/

IK Rig setup
https://docs.unrealengine.com/5.1/en-US/unreal-engine-ik-rig/

Locomotion, Root Movement driven by animation
https://docs.unrealengine.com/5.1/en-US/locomotion-in-unreal-engine/

Motion Warping orientation and position of animation toward a target
https://docs.unrealengine.com/5.1/en-US/motion-warping-in-unreal-engine/

Real-time Machine Learning Deformations through Maya plugin
https://docs.unrealengine.com/5.1/en-US/how-to-use-the-machine-learning-deformer-in-unreal-engine/

IK animation retargeting
https://docs.unrealengine.com/5.1/en-US/runtime-ik-retargeting-in-unreal-engine/

Layered Animation, modular skeletal animation
https://docs.unrealengine.com/5.1/en-US/using-layered-animations-in-unreal-engine/

Sub animation graphs within graphs for greater complexity
https://docs.unrealengine.com/5.1/en-US/using-sub-anim-instances-in-unreal-engine/

Animation Physics Dynamics for secondary motion on solid objects
https://docs.unrealengine.com/5.1/en-US/creating-dynamic-animations-in-unreal-engine/

Ragdoll physics blending per joint for hit effects
https://docs.unrealengine.com/5.1/en-US/physics-driven-animation-in-unreal-engine/

Rigging & Animation Cont.

17) Physics Asset

Physics Assets should only be set up for characters that need a ragdoll or hit effects in the animation blueprint. Depending on the number of joints, this can be a complicated process with mixed results.

Physics Asset Editor
https://docs.unrealengine.com/5.0/en-US/physics-asset-editor-in-unreal-engine/

Examples:
https://www.youtube.com/watch?v=UqJXKFldJSM
https://www.youtube.com/watch?v=OFHxWLRrL5M

a) Approval (by Shannon/John + Matt & Coral)

Material Optimization

Unreal Materials Optimization guide:

https://docs.google.com/document/d/1-guvLUfwk7fcVOuHCTEehJWf7i6AoDoU65jIKwSIG6Q/edit#

For this project, we are using a Master subsurface PBR material M_PBR, which has all the settings needed for most mesh surfaces. This material does not cover VFX materials which have unique setups. Most game materials should use an instance of these master materials.

M_PBR is setup so that there are up to 4 textures needed for a fully textured object:

- Base Color with AO
- Normal with Opacity
- Packed Metal, Rough, Height, Emissive/Subsurface
- optional Specular Color

Feature List

- Triplanar Switch for mapping seamless textures with random variation
- Texture UV Warping, Warp Animation, and UV transforms
- Solid Color parameters with Lerp Alphas
- Min/Max parameters for Metal, Rough, Emissive, etc.
- Subsurface depth controlled through Opacity
- Gradient color overlay on Base Color
- AO multiplies with Base Color
- Height Map blends with Normal Map
- Specular Color through Emissive channel and camera vector
- Fresnel opacity parameters for transparent version
- Material functions re-used to setup Transparent, Landscape, and Decal materials

Material Property Overrides can be used in the Material Instances as needed for changing Blend Mode, Shading Model, Two-Sided, etc.

Default Lit is the default mode, Subsurface should be enabled as needed for the material.

Textures should be assigned to their correct Texture group when uploaded for performance Scaling to work properly.

Normal Map textures that have Opacity should be assigned the Default Compression mode, Normal Map compression ignores the Alpha channel.

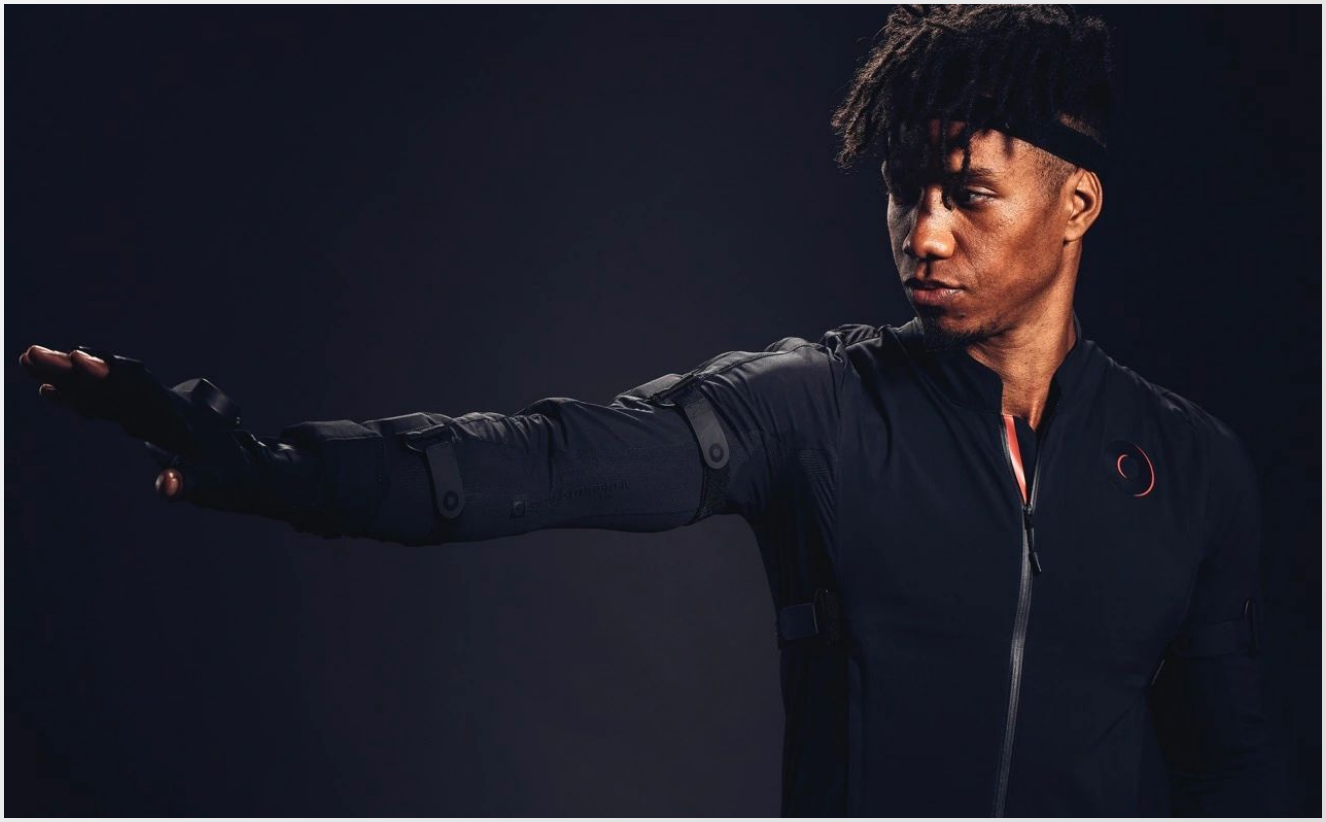
MoCap Information

Unreal Engine 5 Mocap Clean Up Workflow

Rokoko

We can use valve trackers on objects and use constraint systems

Slightly Impeded further information until we are fully aware of all the SCAD has to offer in regards to mo-cap.



Animation Pipeline Guide

[Link To Miro Board]

Approval Processes

During approval steps, your lead will approve or reject your work. If work is accepted, proceed to the next pipeline step. If it is rejected, your lead and your peers will help identify how far the animation needs to move backwards in the pipeline for revisions or be tweaked.

Steps

1. **Create Keyframes**
2. **Get Keyframes Approved**
3. **In Between**
4. **First Pass**
5. **First Pass Approved**
6. **Second Pass and Final**
7. **Final Approval**
8. **Upload to Unreal**

Animating

1. **Create Keyframes**

When creating keyframes, you will need to view references to get your timing correct. Always ask your peers questions and advice if you ever have questions about timing. Make keyframes at all key points of the animation. (ex. A walk cycle will need a keyframe at each time a heel contacts the ground)

2. **Get Keyframes Approved**

Keyframes need to be approved by your lead before continuing to in-betweens.

3. **In Between**

In betweens need to be added once your keyframes are approved. You only need to add enough in betweens to make the animation somewhat smooth until you believe it is ready to be reviewed as a first pass.

4. **First Pass**

Your first pass should be what you believe to be the core animation. There doesn't need to be any polishing or super precise cleanup yet, but there should be enough to get the basic idea of how the animation will work.

5. **First Pass Approval**

First passes need to be approved by your lead before continuing to Second Pass and Final.

6. **Second Pass/Final**

Once your first pass is approved, you are free to move on into the cleanup and polishing stage in order to get a second pass.

7. **Final Approval**

Second passes/Finals should be approved by your lead before continuing to Exporting to Unreal.

8. **Export to Unreal**

If you are animating in Unreal, you won't need to do this step since it will already be in Unreal. If you are exporting to Unreal from Maya or Blender, you will need to make sure animation is named using the correct naming conventions.

Sound Pipeline Guide

Link To Miro Board
Link To Research Doc

Approval Processes

During approval steps, your lead will approve or reject your work. If work is accepted, proceed to the next pipeline step. If it is rejected, your lead and your peers will help identify how far the sound needs to move backwards in the pipeline for revisions or be tweaked.

Request for Sound Effect

For each scrum team, during their sprint they will maintain a spreadsheet for keeping track of sound assets, using the following template:

Game Audio Template

Any team member on the sprint can add to the spreadsheet.

Acquiring Sound Effects

Acquire via Library or Foley

Compiling Sound Effects

Edit in the Reaper DAW.

When exporting, adhere to the standard settings for .wav files for the project.

Archive the edited .wav file in the sound assets folder of the module you are working on.

Naming Convention:
Category_Subcategory_Level_Name_Variation#

Category:
    Overall Type of Sound (SFX, Music, Dialogue)
Subcategory:
    Type of Category (Ambient, Cinematic, Menu, NPC, etc.)
Level:
    What UE Level the sound is located in
Name:
    Identifier of what the sound is
Variation#:
    What variation of the sound it is

    a.) Approval by whoever is in charge of the Test For Doneness of the task. If it is approved, move on to step 4. Otherwise, move backwards in the pipeline to make necessary revisions.

TODO: *Update with more details about file storage*

Implement into Wwise

TODO: *Perforce Details*

- Import wav into appropriate Actor-Mixer Hierarchy
- Edit sounds as necessary
- (Optional - Develop any Game Syncs needed, Game Parameters, States, Syncs, etc.)
- Make Event
- Generate into soundbank.

Import into Unreal

- Go to Window > WAAPI Picker
- Select All > Generate Sound Data

Call sounds in Unreal

https://www.audiokinetic.com/en/library/edge/?source=UE4&id=features_objects.html
https://www.audiokinetic.com/en/library/edge/?source=UE4&id=features_blueprint.html

TODO: *Make video guide for implementation*

Final Approval

Sound Designer and Implementer run the final implemented sound by whoever is in charge of the Test For Doneness of the task.

Poly Counts & Texture Limits

Due to the introduction of Nanite, poly counts differ for three different classes of Objects: Nanite Static Meshes, Static Meshes, and Skeletal Meshes (Characters).

Nanite Static Meshes

Any mesh which can be used with Nanite falls into this category. Nanite does NOT support transparency or any other blend modes besides Opaque and Masked. No polygon limit will exist for Nanite assets. However, assets with many polygons will create large FBXs which will take up group storage space, so only use millions or more polygons in situations where it has a discrete benefit.

Static Meshes

If a static mesh requires transparency or certain other blend modes or rendering features, it cannot use Nanite. In these cases, keep poly count below 20k for non-hero assets, and 100k for hero assets.

Skeletal Meshes (Characters and Morph Targets)

If a skeletal mesh is not a character (ie. exists to make use of Morph Targets, Animations, or other deformations), it follows the same poly limits as Static Meshes.

Due to Nanite, we have a larger budget for characters.

Hero Characters

For Hero characters (the player character, bosses, and other characters who will only ever have one instance of themselves on screen at a time,) the poly limit is 150k.

Non-Hero Characters

For non-hero characters, the poly limit is 50k.

These budgets are set purely for performance. For ease of modeling and rigging, select a poly count which is manageable for the sculptor and rigger. 60-80k for a hero character, and 30k for a non hero character is a recommended target.

To enable these high poly counts, LODs should be created for skeletal mesh characters using Unreal Engine’s auto LOD generation.

Import/Export Rules

Export File Formats

Textures	.tga
3D Assets	.fbx
Sounds	.wav
HDRIs	.hdr
Vector Graphics	.svg

.WAV Files

Adhere to the following settings for all .wav exports:

- 48 Khz
- 16 bit
- Mono for Spatialized Sounds (Footsteps, Attacks, etc.)
- Stereo for Non-Spatialized (Ambients, Weapon Tails, etc.)

Texture Files

- **All textures should be .tga files.**
- On import to unreal, ensure the following:
  - Normal Maps have been automatically converted to “normal map” compression settings and have sRGB unchecked.
  - Value maps or masks have sRGB unchecked. If it isn’t used for a color input, it should not be sRGB.

.FBX Files

All fbx files should be in Unreal Scale, where one unit = 1 cm. This is the scale that should be worked at in maya as well.

In maya, Y is the up axis, and this is true in the FBX coordinate system as well. Unreal engine should be configured to change the up axis to Z while importing 3d assets.

Use the following unreal import settings, unless otherwise needed. Non-specified settings should be left at default unless needed.

Static Meshes (non-deforming):

Name of Setting	***Set To***
Build Nanite	Checked
Generate Missing Collision	Checked
Generate Lightmap UVs	Unchecked
Import Translation/Rotation	0,0,0
Import Uniform Scale	1
Convert Scene	Checked

Leave skeletal-mesh specific options at the default unless you know what you are doing.

Texture Files

- **All textures should be .tga files.**
- On import to unreal, ensure the following:
  - Normal Maps have been automatically converted to “normal map” compression settings and have sRGB unchecked.
  - Value maps or masks have sRGB unchecked. If it isn’t used for a color input, it should not be sRGB.

Narrative Implementation

Notes

Notes will be discoverable in the environment with lore and character information.

Audio Logs

Audio logs add further character and lore context.

Dialogue

Characters will speak with full voice acting and rigged faces

In Engine Cutscenes

Third person cinematics are for big set pieces. They remove control from the player to play a filmic sequence. Cutscenes will be in-engine.

For more minor moments, events will play out with the player still in control, with the normal first person camera.

Project-Wide Functions

The Blueprint Function Library BFL_btbFunctions contains a set of functions which can be used in any blueprint. Some of these functions are simple conveniences: others should be used in place of the default node in all cases. Those ones are marked in **bold**.

btbOpenLevel	Opens a level (by asset reference)
btbIntitalizeLevel	To run after opening a level
btbSetInputMode	Configures input modes between UI and game
btbOpenPauseMenu	Opens the pause menu
btbClosePauseMenu	Closes the pause menu
btbOpenMainMenu	Brings the player to the main menu
btbQuitGame	Exits the game
btbOpenLevelByName	Opens a level (by asset reference)
btbGetGameInstance	Gets the game instance (no need to cast to btb_gameinstance)
btbGetPlayerCharacter	Gets the player character (returns BP_Aster)
btbPreOpenLevel	Ran automatically before opening a level
btbGetMasterGameSeed	Gets the seed for the run

Procedural Systems

Dungeon System

The dungeon system is used for the three subterranean biomes. It functions by creating a grid of cells, then filling that grid with rooms from a list.

Terminology

Cell:
A bounding box which can contain a room

Room:
A single gameplay area which is contained within a cell

Room Level:
The level which contains all the assets and actors for a room

Room Blueprint:
The blueprint which contains all the data about a room

Hall:
A procedural hallway that connects rooms

Hall Connection:
A point in a room from which a hallway can originate

Hall Segment:
Halls are divided into multiple segments, to allow for procedural connections between rooms

Terminology Cont.

Each cell acts as a bounding box for the room. The room can't be bigger than the cell. However, the room should not take on the blocky shape of the cell, but instead be a more organic form fit within it. Similarly, while halls have to meet at hall connection points, consider a "hall" within the cell which leads from that point to a more natural connection.

Halls are used to connect rooms together. Each cell has a set number of connection points, interspersed around its edge. The default configuration is three per side. One connection is at the middle of the room, then two to its left and right. A room can only have one connector per side.

Halls are composed of hall segments. Hall segments are used to automatically connect any two cardinally adjacent rooms, regardless of the positions of their two hall connectors. To connect two connections, three hall segments are used.

Rooms can have up to four connectors, but only one is guaranteed to be filled. If a room has a connector with no adjacent room to connect to, the connector is blocked off by a wall.

The Details

The dungeon system is stored in the ProceduralDungeonSystem module, which is loaded in a plugin. It is managed with a BP_CellDungeonGenerator, which is responsible for creating a grid of BP_Cells, populating that grid with BP_CellRooms, loading in the associated level instances, then connecting them with BP_CellHalls. It contains a weighted list of rooms it can use.

The cell dungeon generator starts by creating a starting room, then “tunneling” to an ending room. Tunneling is creating a path of rooms for a set random length (between a min and max) and then placing the ending room. After tunneling to the end room, offshoot tunnels are made to any number of “mandatory” rooms, such as loot rooms or shops. Finally, it adds rooms to unused adjacent connectors. Tunneling can only be done with rooms with 2 or more connections.

The generator is self-validating. If an operation fails, it retries. If it fails enough times, it restarts the entire generation process. Generation typically happens in a fraction of a second, with instancing in levels taking longer.

Rooms are stored in level files, which are instanced into the level. Each level has an associated blueprint, which is used to store info about the room. The naming convention for rooms is BiomeAbbreviation_RoomName_LVL and BiomeAbbrevation_RoomName_BP. The asset type prefix is instead a suffix that a room level is organized next to its blueprint in the content browser.

Minor Procedural Elements

Enemy placements within a room are randomized, and pulled from a weighted list. Loot is randomized.

Desired and Unimplemented Features

Repeatable seeding for everything
Props and hazards randomized
Randomize some set dressing for variety

Badlands Biome Systems

The badlands biome is primarily handbuilt, with only the minor systems for loot, enemy, and prop placement changin.

Combat

Spawning Enemies

Enemies should be spawned via ***BP_UPDATEthisdocwithname***, which is used for spawning randomized enemies in a room. It has a list of locations in which they can be spawned, and a min/max number to spawn, and picks from a weighted list which can be changed per instance.

Another blueprint, BP_WaveSpawner (unimplemented) will be used to spawn in additional waves of enemies.

Detecting Combat

A system needs to be built to detect if the player is in combat. The suggested method is as follows: when the player deals damage to or takes damage from an enemy, set an “in combat” variable to true, and start a timer. When the timer is finished, run a test to see if any enemies are targeting the player. If they aren’t, end combat. Reset the time when the player deals or takes damage to/from an enemy. **Note: build a unified detection system into btb_enemy**

This system is needed for combat music and other effects which happen based on if combat is happening.

Weapons

Weapons are the main damage-dealing tool of the player. The player has two slots for weapons, which they can switch between at any given time.

Implementation

Weapons are child blueprints of BP_Weapon. BP_Weapon can be picked up, placed in weapon slots, and is set up to receive input from the player via eight event dispatchers: a pressed and released dispatcher for the weapon's primary action, secondary action, melee action, and reload.

By default, BP_Weapon has an implementation of line traced weapons which should be sufficient for any semi-automatic firearm, including audio, particle effects, animation, recoil, and reloading. By changing default variables, the type of gun can be altered.

For more complex behaviors, consider altering certain base functions. By overriding DefaultWeaponFire, a gun could be converted to firing projectiles instead of line traces, while maintaining the rest of the weapon framework's functions.

For more drastic changes, bind entirely new functions to event dispatchers.

Weapon Mods (Unimplemented)

The design accounts for a standard set of modifications that can be applied to any weapon. To implement this system, create a blueprint type for weapon modifications, and the ability for them to be carried by the player. Add the player's carried mods to the save-load system so it persists between levels.
At a workbench, the player can then attach them to a weapon, adding them to an array on the weapon.

The effect of the modification should work on any gun that uses the default functions. If a weapon overrides those functions, implement its effects into the new ones.

Super Bullets (Unimplemented)

Super bullets are a limited ammo type which does greater damage. To activate them hold down reload.

To implement this feature, first add a blueprint for super bullets, and a tracking variable on the player. Add the tracking variable to the save-load system so it persists between levels.

Move reloading from "pressed" to "released", and start a timer when the input is pressed. If the timer is over a certain number, load super bullets instead of normal bullets, if the player is carrying any. Remove that many bullets from the player. If the player has fewer than a full clip, only load that many.

Configuration Variable Description

Variables such as tracking booleans aren't listed; only those which should be changed to configure the effect.

Fire Mode Attributes:

Damage Per Shot (Float)	Damage dealt by one bullet. 100 = normal health bar
Damage Falloff Start (Float)	Distance through a line trace damage falloff begins, 0-1 range where 0 is start of trace, 1 is end.
Range (Float)	Length of trace
Angle Variation (Float)	Angle (in degrees) by which a line trace can randomly deviate.
Number of bullets (float)	Number of lines to trace
PhysicsForce (float)	Amount of force to apply to hit physics objects.
ImpactEmitter (Niagara System)	Particle effect to play at impact location
DamageType (Enum)	Type of damage dealt

Cooldown:

CooldownTime (Float)	Time between shots.

Reload:

ReloadDuration (Float)	How long the reload takes. Also controls the speed of the reload animation.

Buffer:

BufferLength (Float)	How long a fire input remains buffered before it is cleared.

Configuration Variable Description Cont.

Recoil:

RecoilForce (Float)	Force of recoil as applied to cursor.
RecoilPattern(Curve Vector)	Curve asset. Controls recoil pattern.
RecoilReturnSpeed(Float)	Speed at which we “descend” the curve asset
RecoilGainSpeed(Float)	Speed at which we “climb” the curve asset
RecoilTime(float)	Total time recoiling lasts

Animation:

Self describing. Ones not labeled arms apply to the weapon.

Ammo:

Bullets in mag	Number of bullets currently in the magazine
Mag Size	Number of times the gun can be shot without reloading. Note A: this is the number of times it can be fired, not the number of bullets in the mag. A three round burst for example would trace three lines, but consume one round. Note B: Yeah the revolver has a magazine. That’s because this is a metaphor

Weapon Animations:

Weapons have two animations, one played on the player’s arms, another on the gun. All weapon animations should be animated to take one second. The speed the animation is played at is then returned to the original rate by the play rate of the montage. This is so that abilities can affect the speed of reloads.

Gadgets

Gadgets are the primary utility of the player. They serve combat purposes, primarily through movement, damage-dealing, or controlling enemies. The player can hold up to two at a time.

Implementation

Gadgets are children of BP_ActiveAbility. Gadgets receive the inputs “begin execution” and “end execution” from the player, when their button is pressed and released.

Gadgets also have a ST_AsterStatModifiers variable, which can be used to affect aster’s stats, although UpdateStats must be called on BP_Aster for changes to take effect.

Variables in the UI class hold its description.

The ActiveAbility class is lightweight. Most gadget functionality must be built bespokely for the gadget. Many gadgets will likely have other blueprints they spawn, as part of deployables or projectiles.

Aster (Player Character)

As the player character for our game, Aster is a very important blueprint. Aster is handled by BP_Aster. BP_Aster is a child of BP_BtB_Character, which is a child of the engine Character class.

Input Handling

Input handling is currently handled directly on Aster (as opposed to using a player controller). Aster also distributes input to weapons Aster holds.

Movement

Aster currently has the ability to walk, crouch, sprint, and a dodge roll. In addition to refining these behaviors, Slide and Mantle need to be added.

For a slide, check if Aster is sprinting when a crouch input is pressed. If Aster is, reduce the capsule to crouch height, and then have Aster perform a momentum based slide. If Aster jumps, cancel into a jump. If Aster releases crouch, cancel into a walk or run. Block aster from crouching or dodge rolling if Aster is crouching.

Weapons & Gadgets

- Aster has systems to pick up, hold, and utilize 4 weapons. This needs to be reduced to two.
- Aster has systems to pick up and hold 2 gadgets, referred to as Active Abilities in code.

Body Mods

Currently, an implementation for Passive Abilities takes the place of body mods. These need to be overhauled with a body mod system. This includes the implementation of slots, so that Aster can only have one in the torso slot at a time, one in the leg slot at a time, etc. They also need to work with a socket or procedural mesh based system to affect the look of Aster's character model. For some specific body mods, branches need to be added to other sections of Aster's code, to change the behavior of something like a jump.

Stat Changes

By changing the values of the Stat Modifiers variable on the BP_PassiveAbility or BP_ActiveAbility, different stat changes can be applied automatically. Stat changes should always be handled through the Update Stats function, instead of being applied directly, in case multiple sources want to change the same effect.

The Update Stats function gathers a list of all stat modifiers to apply, then the Apply Stats function applies the changes by starting from base values and reapplying all modifiers. ST_AsterStatModifier stores data of stat modifications. To add another source to gather stats from, add it to the Modifiers array in Update Stats.

To add another stat which can be affected, add it to ST_AsterStatModifier, and to the Apply Stats function.

Interaction

On the interaction button being pressed, Aster sends out a line trace, which calls the “Interact” function on BTB_Interactable on whatever it hits.

Aster also traces every frame to the same distance, and displays an interaction prompt if the hit target has a BPC_InteractionPromptComponent on it.

Controller support for interactions needs to be added.

Save/Load

See the save/load document for more details. Aster contains functions to write their state into the game instance, and load from that game instance.

The ability to save/load from file is not yet implemented.

UI

On Begin Play, BP_Aster creates an instance of WBP_HUD and adds it to the viewport.

When the open inventory button is pressed, BP_Aster creates an instance of WBP_Inventory and adds it to the viewport. It is destroyed when it is closed.

Aster calls the library function BtB_OpenPauseMenu when the pause button is pressed.

Health

Healing items can be collected in the dungeon. They are scarce, and using them takes time, like a flask from dark souls. They may be available for purchase in the shop.

Implementation

BP_HealingItem serves as the in-world version of healing items to pick up. Upon being picked up, they are destroyed, and a tracking variable on BP_Aster will keep track of how many Aster is holding. This much is already implemented.

In addition, a trivial implementation of the healing effect has been completed. It needs to be expanded to

A. Block usage of weapons/gadgets while healing
B. Take time to start the heal
C. Apply the healing effect over a short period of time

Shops & Currency

Shops are a currently unimplemented system allowing the player to spend currency to buy items.

Currency

Currency should be kept track of on the player with an integer variable called “ore”. It should exist in the world as BP_Ore. The blueprint contains the model, and the collision volume to automatically pick it up. A function called BtB_DropCurrency can be used to spawn a random amount at a location.

Ore is dropped by enemies on death, and from veins found in walls. These veins will be used with a blueprint called “BP_OreVein”, which on interaction drops a random amount of ore.

The currency variable needs to be able to be passed into and out of the game instance via ST_AsterSaveData.

Shops

The shop blueprint needs to be able to generate a shop in any room it is set up in. It should contain the shopkeeper, and the art asset which is used to display items on sale. The shop blueprint should be able to be dropped into a room level, the location of its models moved, then its loot tables configured. What it specifically sells should be random per instance. The randomness should be based off of the global seed.

Shops sell health items, gadgets, weapon upgrades, and coal, although not at once. For gadgets and weapon upgrades, the shop needs to utilize weighted loot table assets which can be changed out on a per instance basis. A structure data asset which contains the blueprint to sell, the weight (likelyhood to choose) and the price should be created.

When a shop offers gadgets or weapon upgrades, it should always offer multiple options, in order to create meaningful choices.

Input

TBD

TBD

TBD

Saving & Loading

There are two major times the game needs to save and load data: when moving between levels, and when closing and opening the game. There are furthermore two types of data which needs to be saved; single-run data, and metaprogression data.

Saving Between Levels

Single-Run Data:

When moving between levels, all blueprints are destroyed, except for the Game Instance. We use the game instance class BP_BtB_GameInstance to save data between levels. There is a struct asset called ST_AsterSaveData, which aster knows how to read into and write out of to restore important data about Aster's state. When moving between levels, a variable of this structure is passed into the game instance, then from the game instance to the new instance of BP_Aster. Whenever new features are added to Aster, they should be added to ST_AsterSaveData, and BP_Aster's functions to read and write save data.

ST_AsterSaveData is only for single-run data, which can freely be destroyed at the end of a run. Data which progresses between runs, meta progression data, does not belong in this structure.

Meta-Progression Data:

Meta-progression data uses the structure ST_SaveGameData (unimplemented), which is also written into and out of the game instance, as well as written to disk.

Closing/Opening Game

(Unimplemented):
A SaveGame object called BP_BtB_SaveGame will be created to include all data saved to disk.
https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/SaveGame/

Single-Run Data:

To save a significant amount of work, the game cannot save to disk during the middle of a level.

Later on, the ability to save to disk between levels will be added, but only when a good enough system for avoiding cheating death via alt+f4 can be devised. This will be achieved by writing the same ST_AsterSaveData data into the save game.

Meta-progression Data:

The structure ST_MetaSaveData will be created to store all meta-progression data, including statistics (wins/losses, streaks), body modifications which persist, and item unlocks if we reach there. The structure is written into BP_BtB_SaveData when a run ends, and read when the game is opened.

Enemies

Base Classes For Enemies

All enemies must inherit from BP_BtB_Enemy (which inherits from BP_BtB_Character which inherits from Character). They inherit BPI_Damagable from BP_BtB_Character, which allows them to take damage.

Events/functions to override

TakeDamage may be overridden (possibly calling super, or not) if it requires custom behavior. Similarly, Die might be overridden (probably not calling super).

Hitboxes

We are stuck with the capsule collision from Character, which is also used for determining character movement component's movement, but for characters for whom this capsule does not make sense, it should be shrunken, and additional collisions should be added. To be registered as damageable to hitboxes, the components must be tagged as "Damageable")

AI Controllers

AI should use a controller which inherits from BP_BtB_AIController (which inherits from AIController). This includes a handy reference variable to the player character, called “Player Reference” which should be used instead of “get player character”

Behavior Trees

All enemy AI should be programmed using behavior trees.

Model

A controller/puppet model, where the enemy’s blueprint has certain actions (attacks, special movement options, etc) but the ai controller (or the controller’s behavior tree) chooses when and why to use certain actions, is recommended but not required.

Attacks

Attacks should either utilize line traces using the “weapon” trace channel, spawn BP_Hitboxes, or spawn projectiles which have BP_Hitboxes on them. If none of these options are viable, a bespoke solution which calls TakeDamage on the target can be made, but no guarantees will be made that these solutions will be continually supported as changes are made to important blueprints.

BP_hitbox

Spawned with function "spawn and attach hit box" or manually. Attacker is the actor who attacks. The attacker can receive certain notifications via the "bpi_attackcallbacks" interface (not yet implemented). Deals damage of type on overlap. Despawns after specified lifetime. Can be spawned attached to a socket to bind them to the animation.

Behaviors

Enemy behavior should be nuanced enough to be complex and believable. This comes from creating multiple attack patterns, and movement more complex than a single rushdown. Consider using variables like “aggression” or “fear” to make a “mood” for the AI, which are used to make decisions, in addition to other factors like distance from the player.

Enemies should also have a “searching” behavior for when they have seen the player recently, but don’t see them right now, to avoid them jumping right from combat to their idle behavior.

The idle (player not detected) behavior should appear realistic for the enemy type, and might include resting, patrolling, scavenging off corpses, or more.

Bosses

Bosses should have larger health pools than enemies. Bosses might have weak points, but should not follow the “Legend of Zelda” model, where an enemy is invincible until it reveals its weak point during a specific phase. Bosses should have more and more varied attacks, which deal more damage, but are well-telegraphed and avoidable if you know what you’re doing.

Bosses

TBD

TBD

TBD

Graphics Options

Beneath the Badlands will have 3 presets for graphics settings.

RTX/High Quality Mode

RTX mode utilizes hardware ray tracing in conjunction with some lumen features to provide the highest quality look. This mode is the primary target for art direction

Ray Traced AO
Ray Traced Global Illumination
Ray traced reflections w/ screen traces
Ray traced translucency

RTX Settings To Test:
Test brute force vs final gather
Test all samples

Lumen Mode

Lumen mode features more restrained settings, but still uses lumen features to achieve global illumination and remain close to the original look. It exists to help support cards without hardware ray tracing. Depending on performance testing it may still use hardware acceleration for lumen.

Lumen AO
Lumen Global Illumination
Lumen Reflections w/ surface cache
Lumen Translucency

Performance Mode

Performance mode disables features like global illumination, ray traced reflections, ray traced shadows, and ray traced AO. It provides high framerates, at the cost of deviating the furthest from the art direction.

Screen Space Reflections
Screen Space AO
Raster Translucency
SSGI

Individual Settings

To be configured regardless of preset.

Resolution:
Screen Resolution. Max depends on your monitor.

Fullscreen mode:
Fullscreen
Windowed
Borderless

Depth of field:
(if utilized by art team, option to disable)

Anti Aliasing:
Investigate options. Offer as many as available easily in the engine.

Texture Quality:
Caps the texture quality to reduce VRAM usage.

Vsync:
Defaults to off, but can be turned on.

FOV:
Default of 90, but can be changed. Playtest to determine min and max