

**Liverpool John Moores University**

**Department of Computer Science**  
**Final Year Project**

**Project Title:**

Building an interactive application to demonstrate how physics, lighting and effects principles are implemented in modern games engines.

Student Name: Ryan Tinman

Student ID: 723287

Programme Name: Computer Science

Date: 20/04/2018

*This project is submitted for the module 6001PROJ/6000PROJ and complies with all relevant LJMU academic regulations, including plagiarism and collusion.*

## Abstract

This dissertation examines the various ways in which one can learn digital lighting/effects and physics principles through the use of an interactive piece of software. This dissertation will seek to demonstrate how the various principles of lighting/effects and physics - as used in digital media such as videogames and animation, can be used, demonstrated and explained to a user through the use of a software package such as Unity, Unreal or CryEngine.

The main deliverable for this project will be an interactive piece of software that will be used to showcase various physics and lighting/effects principles used in modern video games, animation and other applications to make them seem as lifelike as possible.

The purpose of this software is to demonstrate these principles, and to allow the user to interact with objects in the environment to see what these principles are, how they are simulated and why they are needed.

## Dedication

To my father, who I've always looked up to. For always being there for me when I need help, and for inspiring my love of computers, programming and electronics.

## Acknowledgments

I would like to thank my project supervisor Mr Mike Baskett of the Computer Science department at Liverpool John Moores University for providing the original idea for this project, and helping me evolve it into what it is now. His consistent kind words and assistance helped steer me in the right direction.

I would like to express my profound gratitude to my parents for their unconditional love and continuous support throughout the years, thank you for always being there for me.

Finally, I would like to give a special thank you to my fiancé Michaela, who helped shape me into the person I am today and without whom I would have never met any of my long-time friends, or even went to university. Without you I wouldn't be where I am today.

I love you.

## Table of Contents

Abstract.....	i
Dedication.....	ii
Acknowledgments.....	iii
Table of Contents.....	iv
Table of Figures.....	ix
List of Tables .....	xxi
Definitions and Acronyms.....	xxii
1. Introduction .....	1
1.1. Overview .....	1
1.2. Background .....	1
1.3. Motivation.....	1
1.4. Aims & Objectives .....	2
1.5. Methodology.....	2
1.6. Results of project .....	3
1.7. Roadmap .....	3
2. Problem Analysis & Domain Research.....	5
3. Technical Research.....	6
3.1. Game Engines.....	6
3.1.1. Unity.....	6
3.1.1.1. Pricing.....	6
3.1.1.2. Advantages.....	7
3.1.1.3. Disadvantages .....	7
3.1.2. Unreal.....	7
3.1.2.1. Pricing.....	8
3.1.2.2. Advantages.....	8
3.1.2.3. Disadvantages .....	8
3.1.3. CryEngine .....	9
3.1.3.1. Pricing.....	9
3.1.3.2. Advantages.....	9
3.1.3.3. Disadvantages .....	10
3.1.4. Chosen Engine: Unreal .....	10
3.2. Physics and Lighting/Effects Principles .....	11
3.2.1. Collision Detection .....	11
3.2.2. Cloth Simulation.....	11
3.2.3. Flocking Simulation .....	12

3.2.4. Types of Lights, shadows and Lighting Effects .....	12
3.2.5. Post Processing Effects .....	12
3.2.6. Other Principles.....	13
4. Requirements and Analysis.....	14
4.1. Purpose .....	14
4.2. Scope.....	14
4.3. Product Perspective .....	16
4.3.1. System Interfaces.....	16
4.3.2. User Interfaces .....	16
4.3.3. Hardware Interfaces .....	17
4.3.4. Software Interfaces.....	17
4.3.5. Communications Interfaces .....	17
4.3.6. Memory.....	17
4.3.7. Operations .....	17
4.3.8. Site Adaption Requirements.....	18
4.4. Product Functions .....	18
4.5. User Characteristics .....	18
4.6. Limitations.....	19
4.7. Assumptions and Dependencies.....	19
4.8. Apportioning of Requirements .....	19
4.9. Specific Requirements .....	20
4.10. Usability Requirements.....	20
4.11. Performance Requirements.....	20
4.12. Design constraints.....	20
4.13. Standards compliance.....	20
4.14. Software System Attributes .....	21
4.14.1. Reliability.....	21
4.14.2. Availability.....	21
4.14.3. Security .....	21
4.14.4. Maintainability .....	21
4.14.5. Portability.....	21
4.15. Verification.....	21
5. Design.....	22
5.1. Overview .....	22
5.1.1 Purpose .....	22
5.1.2 Background .....	22

5.2. Features .....	22
5.2.1 General Features.....	22
5.2.2 Gameplay .....	22
5.3. Game World.....	23
5.3.1 Overview.....	23
5.3.2 Camera .....	23
5.3.3 Game Engine .....	23
5.3.4 Environment.....	24
5.4. User Interface .....	24
5.4.1 Overview .....	24
5.5. Level Designs.....	25
5.5.1. Starting Zone .....	25
5.5.2. Physics Area .....	25
5.5.2.1. Collision Demonstration .....	26
5.5.2.2. Cloth Demonstration.....	28
5.5.2.3. Molecular Simulation Demonstration.....	30
5.5.2.4. Flocking Demonstration .....	32
5.5.3. Lighting and Effects Area .....	33
5.5.3.1. Types of Light, Shadow, and Lighting Effects Demonstration .....	33
5.5.3.2. Post Processing Effects Demonstration .....	34
5.5.3.3. Materials and Material Properties Demonstration .....	36
5.5.3.4. Reflections Demonstration .....	37
5.5.4. Sandbox Area .....	37
6. Implementation .....	38
6.1. Project Creation, Landscape & Whiteboxing .....	38
6.2. Collision Detection Building (Physics Area).....	77
6.3. Cloth Simulation Building (Physics Area) .....	118
6.4. Flocking Simulation Area (Physics Area) .....	229
6.5. Level Changes & Finishing Touches .....	262
6.6. Projectiles.....	283
6.7. Types of Light Building (Lighting Area).....	303
6.7.1. Point Light Demo.....	303
6.7.2. Spot Light Demo.....	316
6.7.3. Directional Light Demo .....	324
6.8. Post Processing Effects Building (Lighting Area) .....	332
6.8.1. Anti-Aliasing Demo.....	334

6.8.2. Depth of Field Demo .....	336
6.8.3. Vignette Demo .....	339
6.8.4. Lens Flare Demo.....	341
6.8.5. Bloom Demo .....	342
6.8.6. Eye Adaption Demo .....	345
7. Testing.....	348
7.1. Application Testing.....	348
7.2. Peer Testing .....	348
8. Evaluations and Conclusions.....	350
9. References .....	353
10. Appendices.....	355
10.1. Monthly Meeting Forms .....	355
10.1.1. November .....	355
10.1.2. December.....	356
10.1.3. January.....	357
10.1.4. February.....	358
10.1.5. March .....	359
10.2. Gantt Chart.....	360
10.3. Peer Testing Questionnaire .....	361
10.4. Testing Log .....	362
10.5. Source Code Listing.....	366
Collision Demo .....	366
CollisionDemo.h .....	366
CollisionDemo.cpp .....	369
Flocking .....	380
FlockingAI.h.....	380
FlockingAI.cpp .....	384
Lighting.....	395
LightDemo.h.....	395
LightDemo.cpp .....	397
Miscellaneous .....	402
MyStaticMesh.h .....	402
MyStaticMesh.cpp .....	403
TextContainerComponent.h .....	404
TextContainerComponent.cpp.....	405
Molecular Simulation.....	406

MolecularSim.h .....	406
MolecularSim.cpp .....	409
Physics.....	418
RopeSim.h .....	418
RopeSim.cpp .....	419
VerletSim.h.....	428
VerletSim.cpp.....	434
ClothSim.h.....	443
ClothSim.cpp .....	444
Projectiles.....	458
StickProjectile.h .....	458
StickProjectile.cpp.....	459
ProjectileVolume.h.....	459
ProjectileVolume.cpp.....	460
Wind.....	463
WindSim.h.....	463
WindSim.cpp .....	464
Modified Default Project Code.....	466
FinalProjectRyanTHUD.h.....	466
FinalProjectRyanTHUD.cpp .....	467
FinalProjectRyanTProjectile.h .....	471
FinalProjectRyanTProjectile.cpp .....	472

## Table of Figures

Figure 1 - Unity 3D Interface .....	6
Figure 2 - Unreal Interface .....	7
Figure 3 - CryEngine Interface.....	9
Figure 4 - World Overview .....	23
Figure 5 - User Interface .....	24
Figure 6 - Starting Zone.....	25
Figure 7 - Physics Area Overview .....	25
Figure 8 - Collision Building.....	26
Figure 9 - Collision Demonstration .....	27
Figure 10 - Cloth Building.....	28
Figure 11 - Cloth Demonstration .....	29
Figure 12 - Molecular Simulation Building.....	30
Figure 13 - Molecular Simulation Demonstration .....	31
Figure 14 - Flocking Area.....	32
Figure 15 - Lighting/Effects Area Overview .....	33
Figure 16 - Types of Light Building .....	33
Figure 17 - Post Processing Building .....	34
Figure 18 - Post Processing Demonstrations .....	35
Figure 19 - Materials Building .....	36
Figure 20 - Project Creation .....	38
Figure 21 - Default Level .....	39
Figure 22 - New Level Dialogue.....	39
Figure 23 - Blank Level .....	40
Figure 24 - Landscape Editor.....	40
Figure 25 - Created Landscape.....	41
Figure 26 - Lightmass Importance Volume .....	42
Figure 27 - Start of Whiteboxing.....	42
Figure 28 – Brush Settings .....	43
Figure 29 - Whiteboxing Starting Zone Complete.....	43
Figure 30 – Placing Geometry.....	44
Figure 31 - Brush Settings .....	44
Figure 32 - Subtracted Geometry .....	45
Figure 33 - Start of Landscape Sculpting.....	45
Figure 34 - Starting to Make Mountains .....	46
Figure 35 - Rough Mountains Progress.....	46
Figure 36 - Sculpting Progress Overview .....	47
Figure 37 - Before Smooth Tool .....	47
Figure 38 - Smooth Tool After.....	48
Figure 39 - Erode Tool Before .....	48
Figure 40 - Erode Tool After .....	49
Figure 41 - Erode Tool Before .....	49
Figure 42 - Erode Tool After .....	50
Figure 43 - Stretched Textures.....	50
Figure 44 - Retopologise Tool .....	51
Figure 45 - Whiteboxed Level Areas .....	51
Figure 46 - New Material .....	52

Figure 47 - New Material in Content Browser .....	52
Figure 48 - The Material Editor .....	53
Figure 49 - First Texture Added.....	53
Figure 50 .....	54
Figure 51 .....	55
Figure 52 .....	55
Figure 53 .....	56
Figure 54 .....	56
Figure 55 .....	57
Figure 56 .....	58
Figure 57 - Adding New LandscapeLayerBlend.....	58
Figure 58 .....	59
Figure 59 .....	59
Figure 60 .....	59
Figure 61 .....	60
Figure 62 .....	60
Figure 63 .....	60
Figure 64 .....	61
Figure 65 .....	61
Figure 66 .....	62
Figure 67 .....	62
Figure 68 .....	63
Figure 69 .....	63
Figure 70 .....	63
Figure 71 .....	64
Figure 72 .....	65
Figure 73 – Landscape Texture Tiling.....	65
Figure 74 - Changing Mapping Scale to 4.5.....	66
Figure 75 - Landscape Texture Tiling Fixed.....	66
Figure 76 - Texturing Starting Zone .....	67
Figure 77 - Texturing Complete .....	67
Figure 78 - Texture Blending.....	68
Figure 79 - Texture Blending.....	68
Figure 80 .....	69
Figure 81 - Texture Blending.....	69
Figure 82 .....	70
Figure 83 - Overview of Texture Blending .....	70
Figure 84 .....	71
Figure 85 – Path Created with Spline Tool.....	71
Figure 86 - Changing Path Width .....	72
Figure 87 .....	72
Figure 88 .....	73
Figure 89 .....	73
Figure 90 - Mesh Added to Path .....	74
Figure 91 .....	74
Figure 92 - Completed Starting Zone with Paths .....	75
Figure 93 - Sub-Level Creation .....	75
Figure 94 - Three Created Sub-Levels .....	75

Figure 95 .....	76
Figure 96 .....	76
Figure 97 .....	77
Figure 98 .....	77
Figure 99 .....	78
Figure 100 .....	78
Figure 101 .....	79
Figure 102 - Position of Building in Physics Area .....	79
Figure 103 - Whiteboxed Building.....	80
Figure 104 .....	80
Figure 105 - Ceiling Light.....	81
Figure 106 .....	81
Figure 107 .....	82
Figure 108 .....	82
Figure 109 .....	83
Figure 110 .....	83
Figure 111 .....	84
Figure 112 - Collision Building Top-Down View .....	84
Figure 113 - New Actor Dialogue .....	85
Figure 114 - Initial Code .h File.....	85
Figure 115 - Initial Code .cpp File.....	86
Figure 116 .....	86
Figure 117 .....	87
Figure 118 - UpdateWorldSpaceVerts .....	87
Figure 119 – DrawTriangle .....	87
Figure 120 - Triangle Rendered In-Game .....	88
Figure 121 - AddVert & ClearVerts.....	88
Figure 122 – New Functions Being Called.....	89
Figure 123 - UpdatePlanes.....	89
Figure 124 - DrawDebugPlanes.....	89
Figure 125 - Gameplay Screenshot .....	90
Figure 126 .....	90
Figure 127 - PerformCollisionCheck.....	90
Figure 128 .....	91
Figure 129 .....	91
Figure 130 - Progress So Far .....	91
Figure 131 .....	92
Figure 132 - Adding Text To Component .....	92
Figure 133 - Text Rendered In-Game .....	93
Figure 134 - More Text Rendered In-Game .....	93
Figure 135 .....	94
Figure 136 - Gameplay Screenshot .....	94
Figure 137 .....	95
Figure 138 - Gameplay Screenshot .....	95
Figure 139 .....	95
Figure 140 .....	96
Figure 141 .....	96
Figure 142 .....	96

Figure 143 .....	96
Figure 144 .....	96
Figure 145 .....	97
Figure 146 .....	97
Figure 147 - Level Sequencer .....	98
Figure 148 - Adding 'Collision Demo' To Sequence .....	99
Figure 149 - Finished Collision Demo Level Sequence .....	99
Figure 150 - Level Sequence: Camera Cuts .....	100
Figure 151 - Level Sequence: Collision Demo .....	100
Figure 152 - Collision Demo: Right Text .....	101
Figure 153 - Collision Demo: World Space Line Offset .....	101
Figure 154 - Collision Demo: Collision Render Mode .....	102
Figure 155 - Collision Demo: Face Plane Alpha .....	102
Figure 156 - Collision Demo: Render Intersect .....	103
Figure 157 - Collision Demo: Text Container Index .....	104
Figure 158 - Collision Demo: Level Sequence Pause Time .....	104
Figure 159 - Collision Demo: Render All Edge Planes .....	105
Figure 160 - Collision Demo: Render Line .....	105
Figure 161 - Creating TextContainerComponent Class .....	106
Figure 162 - Creating TextContainerComponent Class .....	106
Figure 163 - Initial Code TextContainerComponent Class .h .....	107
Figure 164 - Initial Code TextContainerComponent Class .cpp .....	107
Figure 165 .....	107
Figure 166 .....	108
Figure 167 .....	108
Figure 168 .....	109
Figure 169 .....	109
Figure 170 .....	109
Figure 171 .....	110
Figure 172 .....	110
Figure 173 .....	110
Figure 174 .....	111
Figure 175 .....	111
Figure 176 .....	111
Figure 177 .....	112
Figure 178 .....	112
Figure 179 .....	113
Figure 180 .....	114
Figure 181 .....	114
Figure 182 .....	115
Figure 183 .....	115
Figure 184 .....	116
Figure 185 .....	116
Figure 186 .....	116
Figure 187 .....	116
Figure 188 .....	117
Figure 189 – Project Settings (3 Images) .....	119
Figure 190 .....	120

Figure 191 .....	121
Figure 192 .....	121
Figure 193 .....	122
Figure 194 .....	122
Figure 195 .....	123
Figure 196 .....	124
Figure 197 .....	125
Figure 198 .....	125
Figure 199 .....	126
Figure 200 .....	126
Figure 201 .....	127
Figure 202 .....	127
Figure 203 – Adding a New Class (Three Images) .....	128
Figure 204 .....	129
Figure 205 .....	129
Figure 206 .....	130
Figure 207 .....	130
Figure 208 .....	131
Figure 209 .....	132
Figure 210 .....	132
Figure 211 .....	133
Figure 212 .....	133
Figure 213 .....	134
Figure 214 .....	134
Figure 215 .....	135
Figure 216 .....	135
Figure 217 .....	136
Figure 218 .....	136
Figure 219 .....	137
Figure 220 – VerletSim (Part 1) .....	138
Figure 221 – VerletSim (Part 2) .....	139
Figure 222 .....	140
Figure 223 .....	140
Figure 224 – RopeSim (Part 1) .....	140
Figure 225 – RopeSim (Part 2) .....	141
Figure 226 – Making New Class .....	142
Figure 227 .....	143
Figure 228 .....	143
Figure 229 .....	144
Figure 230 .....	145
Figure 231 .....	146
Figure 232 .....	147
Figure 233 .....	148
Figure 234 .....	149
Figure 235 .....	149
Figure 236 .....	150
Figure 237 .....	151
Figure 238 .....	152

Figure 239 – ClothSim (Three Images) .....	155
Figure 240 – Rope & Cloth .....	156
Figure 241 .....	157
Figure 242 .....	157
Figure 243 .....	158
Figure 244 .....	158
Figure 245 – Build Mesh (5 Images).....	162
Figure 246 .....	163
Figure 247 .....	163
Figure 248 .....	164
Figure 249 – Fixing Rope (4 Images) .....	165
Figure 250 – Adding to VerletSim (3 Images) .....	166
Figure 251 .....	167
Figure 252 .....	167
Figure 253 .....	168
Figure 254 .....	168
Figure 255 .....	169
Figure 256 .....	170
Figure 257 .....	170
Figure 258 .....	171
Figure 259 .....	171
Figure 260 .....	172
Figure 261 .....	172
Figure 262 .....	173
Figure 263 .....	173
Figure 264 .....	174
Figure 265 .....	174
Figure 266 .....	174
Figure 267 .....	175
Figure 268 .....	175
Figure 269 – Making New Actor (4 Images).....	177
Figure 270 – Adding Functions (2 Images).....	179
Figure 271 – Additions to VerletSim (2 Images) .....	179
Figure 272 .....	180
Figure 273 .....	181
Figure 274 .....	182
Figure 275 .....	183
Figure 276 .....	184
Figure 277 .....	184
Figure 278 .....	185
Figure 279 .....	185
Figure 280 – Debug Draw Functions(3 Images) .....	188
Figure 281 .....	188
Figure 282 .....	189
Figure 283 .....	189
Figure 284 .....	190
Figure 285 .....	190
Figure 286 .....	191

Figure 287 .....	192
Figure 288 .....	192
Figure 289 .....	193
Figure 290 .....	193
Figure 291 .....	194
Figure 292 .....	194
Figure 293 .....	195
Figure 294 .....	195
Figure 295 .....	196
Figure 296 .....	196
Figure 297 .....	197
Figure 298 .....	197
Figure 299 .....	198
Figure 300 .....	199
Figure 301 .....	199
Figure 302 .....	200
Figure 303 .....	200
Figure 304 .....	201
Figure 305 .....	202
Figure 306 .....	202
Figure 307 .....	203
Figure 308 .....	203
Figure 309 .....	204
Figure 310 .....	205
Figure 311 .....	205
Figure 312 .....	206
Figure 313 .....	206
Figure 314 .....	207
Figure 315 .....	207
Figure 316 .....	208
Figure 317 .....	208
Figure 318 .....	209
Figure 319 .....	209
Figure 320 .....	210
Figure 321 .....	211
Figure 322 .....	211
Figure 323 – Completed Cloth Demonstration (5 Images) .....	214
Figure 324 .....	214
Figure 325 .....	215
Figure 326 .....	215
Figure 327 .....	216
Figure 328 .....	217
Figure 329 .....	217
Figure 330 – Finished Sequence 5 (4 Images) .....	219
Figure 331 .....	220
Figure 332 .....	221
Figure 333 – Sequence 6 .....	221
Figure 334 .....	222

Figure 335 .....	222
Figure 336 .....	223
Figure 337 .....	223
Figure 338 .....	224
Figure 339 .....	224
Figure 340 .....	225
Figure 341 .....	225
Figure 342 – Rope Demonstration (4 Images) .....	227
Figure 343 .....	228
Figure 344 .....	228
Figure 345 .....	228
Figure 346 .....	229
Figure 347 .....	229
Figure 348 .....	230
Figure 349g .....	230
Figure 350 .....	231
Figure 351 .....	231
Figure 352 .....	232
Figure 353 .....	232
Figure 354 .....	232
Figure 355 .....	233
Figure 356 .....	233
Figure 357 .....	234
Figure 358 .....	234
Figure 359 .....	234
Figure 360 .....	235
Figure 361 .....	235
Figure 362 .....	235
Figure 363 .....	236
Figure 364 .....	236
Figure 365 .....	237
Figure 366 .....	237
Figure 367 .....	238
Figure 368 .....	238
Figure 369 .....	239
Figure 370 .....	239
Figure 371 .....	240
Figure 372 .....	240
Figure 373 .....	241
Figure 374 .....	241
Figure 375 .....	242
Figure 376 .....	242
Figure 377 .....	243
Figure 378 .....	243
Figure 379 .....	244
Figure 380 .....	244
Figure 381 .....	245
Figure 382 .....	245

Figure 383 .....	246
Figure 384 .....	246
Figure 385 – Level Sequence Code (4 Images).....	248
Figure 386 .....	248
Figure 387 .....	249
Figure 388 .....	249
Figure 389 .....	249
Figure 390 .....	250
Figure 391 .....	250
Figure 392 .....	251
Figure 393 .....	252
Figure 394 .....	252
Figure 395 .....	253
Figure 396 .....	254
Figure 397 .....	254
Figure 398 .....	254
Figure 399 – Flocking Demonstration (16 Images).....	262
Figure 400 .....	262
Figure 401 .....	263
Figure 402 .....	263
Figure 403 .....	264
Figure 404 .....	264
Figure 405 .....	265
Figure 406 .....	265
Figure 407 .....	266
Figure 408 .....	267
Figure 409 .....	267
Figure 410 .....	268
Figure 411 .....	268
Figure 412 .....	269
Figure 413 .....	269
Figure 414 .....	270
Figure 415 .....	270
Figure 416 .....	271
Figure 417 .....	271
Figure 418 .....	272
Figure 419 .....	272
Figure 420 .....	273
Figure 421 .....	273
Figure 422 .....	274
Figure 423 .....	274
Figure 424 .....	275
Figure 425 .....	275
Figure 426 .....	276
Figure 427 .....	276
Figure 428 .....	277
Figure 429 .....	278
Figure 430 .....	279

Figure 431 .....	280
Figure 432 .....	280
Figure 433 .....	281
Figure 434 .....	281
Figure 435 .....	282
Figure 436 .....	282
Figure 437 .....	283
Figure 438 .....	283
Figure 439 .....	284
Figure 440 .....	284
Figure 441 .....	284
Figure 442 .....	285
Figure 443 .....	285
Figure 444 .....	286
Figure 445 .....	286
Figure 446 .....	286
Figure 447 .....	287
Figure 448 .....	288
Figure 449 .....	289
Figure 450 .....	289
Figure 451 .....	290
Figure 452 .....	290
Figure 453 .....	291
Figure 454 .....	291
Figure 455 .....	292
Figure 456 .....	292
Figure 457 .....	293
Figure 458 .....	293
Figure 459 .....	294
Figure 460 .....	294
Figure 461 .....	295
Figure 462 .....	296
Figure 463 .....	297
Figure 464 .....	297
Figure 465 .....	298
Figure 466 .....	298
Figure 467 .....	298
Figure 468 .....	299
Figure 469 .....	299
Figure 470 .....	300
Figure 471 .....	300
Figure 472 .....	301
Figure 473 .....	301
Figure 474 .....	302
Figure 475 .....	302
Figure 476 .....	303
Figure 477 .....	304
Figure 478 .....	305

Figure 479 .....	305
Figure 480 .....	306
Figure 481 .....	306
Figure 482 .....	306
Figure 483 .....	307
Figure 484 .....	307
Figure 485 .....	307
Figure 486 .....	308
Figure 487 .....	309
Figure 488 .....	309
Figure 489 .....	309
Figure 490 .....	310
Figure 491 .....	310
Figure 492 .....	311
Figure 493 .....	311
Figure 494 .....	312
Figure 495 .....	312
Figure 496 .....	313
Figure 497 .....	313
Figure 498 .....	314
Figure 499 .....	314
Figure 500 .....	315
Figure 501 .....	316
Figure 502 .....	316
Figure 503 .....	317
Figure 504 .....	317
Figure 505 .....	318
Figure 506 .....	318
Figure 507 .....	319
Figure 508 .....	319
Figure 509 .....	320
Figure 510 .....	320
Figure 511 .....	321
Figure 512 .....	321
Figure 513 .....	322
Figure 514 .....	322
Figure 515 – Spotlight Movement (2 Images).....	323
Figure 516 .....	324
Figure 517 .....	324
Figure 518 .....	325
Figure 519 .....	325
Figure 520 .....	326
Figure 521 .....	326
Figure 522 .....	327
Figure 523 .....	327
Figure 524 .....	328
Figure 525 .....	329
Figure 526 .....	329

Figure 527 .....	330
Figure 528 – Toggling Shadows (2 Images).....	331
Figure 529 .....	331
Figure 530 .....	332
Figure 531 .....	332
Figure 532 .....	333
Figure 533 .....	333
Figure 534 .....	334
Figure 535 .....	334
Figure 536 .....	335
Figure 537 .....	335
Figure 538 .....	336
Figure 539 .....	336
Figure 540 .....	337
Figure 541 .....	337
Figure 542 .....	338
Figure 543 .....	338
Figure 544 .....	339
Figure 545 .....	339
Figure 546 .....	340
Figure 547 .....	340
Figure 548 .....	341
Figure 549 .....	341
Figure 550 .....	342
Figure 551 .....	342
Figure 552 .....	343
Figure 553 .....	343
Figure 554 .....	344
Figure 555 .....	344
Figure 556 .....	345
Figure 557 .....	345
Figure 558 .....	346
Figure 559 .....	346
Figure 560 .....	347

## List of Tables

Table 1 - Acronyms.....	xxiii
Table 2 – Planned Features.....	22
Table 3 - Peer Testing Results .....	349

## Definitions and Acronyms

Acronym/Phrase	Definition
2D	2 dimensional
3D	3 dimensional
Android	A mobile operating system
API	Application programming Interface
Direct3D	A Microsoft API used for 2d and 3d rendering
Dof	Depth of field
HUD	Heads up Display
Linux	An open source operating system
Mac	Short for Macintosh. Apple Computers
OS X	A Macintosh operating system
OpenGL	An API used for 2d and 3d rendering
OpenGL ES	An API used for 2d and 3d rendering (Mobile version)
PC	Personal Computer
PS-Vita	A portable Sony PlayStation Games console
PlayStation 3	A Sony PlayStation Games console
PlayStation 4	A Sony PlayStation Games console
Sprite sheets	An asset similar to a texture which contains many 2d images
Starting zone	The area in the application where the player starts in the world
The application	Refers to this projects software
The player	Refers to the person using the application
UE4	Unreal Engine version 4
UI	User interface
UV/ UV Co-ordinates	2D co-ordinates used to lookup a texture
VR	Virtual reality
Web browsers	Software used the navigate the internet
Wii U	A Nintendo Games console
Windows	A Microsoft operating system
Windows Phone	A mobile phone using the windows operating system
Xbox One	A Microsoft Games console
Xbox360	A Microsoft Games console
animation	The process of moving a 3d model using an asset to define natural movements.
Asset	Describes data content stored in files for the application such as models, textures and sounds
billboard	A 2d primitive in the 3d world which always faces the camera
demonstration areas	Describes the collection of areas in the application
digital learning applications	Educational software
game engine	A development environment used for the creation of video games
iOS	An Apple operating system for mobile phones and tablets
level/levels	Describes the world in the application
lighting area	The area in the application that demonstrated lighting
Mesh	Another term for a 3d model
modelling/3D modelling	The process of authoring 3d models using software
multimedia applications	An application that uses multiple media types such as sound, text, video and animations
node-based	A representation of hierarchical data
physics area	The area in the application that demonstrated physics

player model	The 3d model that represents the user in the world
primitive objects	Describes simple geometry models like cubes and spheres
Procedural	Describes data that is generated by an algorithm
sandbox area	An informal mix of elements from all of the demo areas
Scripting	Predefined actions stored in an asset
Shaders	The programs use by the GPU to render graphics
source code	Describes the text that is the applications code
Spawn	A term used to describe the creating of an object in a applications world
Sprites	A 2d primitive used to draw an image
tech demo	A demonstration of Hardware or software techniques
Users	The people that use the application
white-box	Describes a level is very basic form, no textures etc.
Workflow	Describes the stages of process or task from start to finish
CPU	Central processing unit
GPU	Graphics processing unit

Table 1 - Acronyms

## 1. Introduction

### 1.1. Overview

In recent years there has been an enormous growth in interactive experiences and digital learning applications. These multimedia applications provide various ways to teach people a subject; not just in the traditional sense of reading about a subject and answering questions. They allow users to be a part of the learning process, experiencing first-hand the thing in which they are learning, and providing a way to interact with the subject matter.

An example of this would be the application “Rosetta Stone”; it aims to teach the user a new language. It aims to do this in the same way one would learn their first language as a child. Rather than reading new words and assigning them to words in the user’s known language, it speaks the new words to the user, relating them to images or scenarios. The application has various ways of making sure the user has learnt the new words, such as speaking various words to the user, and presenting them images, the user will then select the image they think corresponds to the spoken phrase (For example the application will say “*Un Uomo*” and the user must select the image of “A man”). Through this interactive experience, users will learn a new language as if they were learning their first.

The traditional forms of teaching, such as lecturing, have been used for hundreds of years. But, there are numerous concerns regarding the effectiveness of these teaching methods, and the engagement of the students attending the lectures. This is because studying notes from lectures is considered by some to be an ineffective way of learning. By using a game engine, a rich, interactive application can be created, rather than the more traditional question and answer revision-style learning applications.

### 1.2. Background

Previously, game engines were not available to the public, and in order for someone not working in the games industry to develop a game or piece of interactive software, they would have to make their own engine. This makes it a very niche area, making it very difficult for someone without previous experience in development to make the piece of software they want. But now more than ever, with the increasing popularity of open to the public game engines such as Unity, it is very easy for anyone with the right motivation to make a professional interactive experience. This is because these software packages are very heavily documented, and there is a wealth of online tutorials and forums. Making it very accessible, and providing all the necessary resources for anyone with an idea to realise it.

A game engine is a development environment used for the creation of video games. Game engines are made up of multiple components working together to create the full application, most engines include at least the following core functionality: A rendering engine to display graphics (Either 2D or 3D), a physics engine to control objects in the world, a sound engine to play sounds and a scripting environment to code logic behind objects. Some game engines may also have additional features such as networking, animation or artificial intelligence.

### 1.3. Motivation

Lots of people find traditional teaching methods ‘boring’, and can find it difficult to remain focused and stay concentrated on the lecturer after long periods of time. Considering the fact that students have to make notes on these lectures, if they lose focus, their notes will have gaps in them, making learning and revising the subject matter more difficult. In addition to this, students with learning

disabilities or disorders such as dyslexia or ADHD will find it difficult to concentrate, follow, or understand lecture material first time around.

Many want to depart from the conventional methods of teaching, and move to interactive computer learning applications that enable students to gain a comprehensive understanding of a topic through hands-on learning. These multimedia applications are not bound by the same limitations as traditional teaching methods. Interactive learning applications open up new worlds of teaching.

#### 1.4. Aims & Objectives

This dissertation examines the various ways in which one can learn digital lighting/effects and physics principles through the use of an interactive piece of software. This dissertation will seek to demonstrate how the various principles of lighting/effects and physics - as used in digital media such as videogames and animation, can be used, demonstrated and explained to a user through the use of a software package such as Unity, Unreal or CryEngine.

The main deliverable for this project will be an interactive piece of software that will be used to showcase various physics and lighting/effects principles used in modern video games, animation and other applications to make them seem as lifelike as possible.

The purpose of this software is to demonstrate these principles, and to allow the user to interact with objects in the environment to see what these principles are, how they are simulated and why they are needed.

The software could be used as a teaching tool, to demonstrate what modern game engines are capable of, and to teach people how various things are implemented within them. Additionally, showing aspiring games development students considering the field what things go in to a game or simulation to make it look as life-like and realistic as possible, and what kinds of things they may be studying at university. The application will be made in Unreal Engine 4.0 using the standard library of assets provided with the program, along with possible other resources that may be deemed necessary later in development.

#### 1.5. Methodology

The interactive application will be developed using Unreal Engine 4.0, details about the design for this application are detailed in chapter 5 of this document. Once the application is in a completed state, it will be tested for errors, and any issues fixed if necessary. A comprehensive, in-depth walkthrough of the development of the application can be found in chapter 6 of this document, and a full listing of the source code in appendix 10.5.

Once fully complete, two groups of users will be selected; one group will use the application, and the other will be given a lecture to take.

Those using the application will visit each of the areas and partake in the demonstrations and interactive portions of the application, the purpose of which will be to teach them the basis of the techniques demonstrated.

Those taking a traditional lecture will have lecture material delivered to them, this material will cover everything that the application does, but without users being able to see what happens, or interact with it in any way. This will essentially be a written-down version of the demonstrations in the application.

The two groups of people will then be given an identical questionnaire, in which they will be asked to answer a range of questions about various topics delivered to them either in the application or

the lecture. The results of these questionnaires will then be compared to judge how effective the application is at teaching users a subject in comparison with a more ‘analogue’ form of learning.

### [1.6. Results of project](#)

From the peer testing of the developed application, it was discovered that, on average, people who used the interactive application to learn the presented techniques, had a deeper understanding of them than those who learned the same techniques from reading a traditional lecture and supplementary notes.

This was tested through the use of a questionnaire, in which people were asked to answer questions about the techniques that were demonstrated or taught to them.

However, the sample size was extremely small for this test. It was intended to be more of a test of how effective the developed application is in practice, and how comparable it is to a form of ‘analogue’ teaching, rather than how much more effective digital learning is to traditional lecture style learning.

Nevertheless, interacting with a topic, being fully immersed in it, and being able to physically see how it works gives people a fuller understanding of a topic than simply being told how it works.

### [1.7. Roadmap](#)

#### Chapter 1 – Introduction

This chapter contains an introduction and overview of the whole project.

#### Chapter 2 – Problem Analysis & Domain Research

This chapter contains an analysis of the problems introduced in the ‘Motivation’ section of the introduction, and research into this domain.

#### Chapter 3 – Technical Research

This chapter contains technical research important to the development of the application planned for this project.

#### Chapter 4 – Requirements and Analysis

This chapter contains a software requirements specification for the application planned for this project. It follows the standards set by the “BS ISO/IEC/IEEE 29148:2011 Systems and Software Engineering - Life Cycle Processes - Requirements Engineering” document, available from the IEEE Xplore Digital Library.

#### Chapter 5 – Design

This chapter contains a design document for the application, it includes information about how the application will act, as well as designs for its various aspects.

#### Chapter 6 – Implementation

This chapter contains what is essentially a step-by-step walkthrough of the development of the application. This chapter shows the development process from start to finish, since those reading this report may not have access to the application, it also contains screenshots of each demonstration.

## Chapter 7 – Testing

This chapter contains the testing results for the application, and the results of the peer testing that was performed on the application.

## Chapter 8 – Evaluations and Conclusions

This chapter contains an evaluation of the software product and the project as a whole.

## Chapter 9 – References

This chapter contains all reference material used in this project.

## Chapter 10 – Appendices

This chapter contains any appendices, such as large tables, source code listing etc.

## 2. Problem Analysis & Domain Research

One of the main purposes of schools and universities is to educate students on essential topics that will be of importance to them later in life; be it to prepare for employment, or to teach life skills such as critical thinking, effective analysing skills, or developing a strong work ethic.

Depending on the university, almost all content is taught through traditional lectures that typically last in the range of 1-3 hours. These lectures should be very well written and presented to give students the opportunity to make effective notes and gain a good understanding of the topic or subject.

However, many universities face a wide range of problems when it comes to student engagement. Numerous students face problems when trying to stay focused on lectures; losing concentration, or even simply finding lecture material boring. It is often said that student's attention span lasts for around 10-15 minutes, but the evidence to support this is lacking. There is however evidence to suggest that students don't lose focus after 15 minutes of a lecture, but their focus gradually wanes over time. Students lose focus or wander off briefly for a minute or two periodically, each time the duration of the loss of focus increasing [1]. Many students also blame lack of sleep for their inability to concentrate during lectures, some even missing lectures entirely. Further to not being able to concentrate, some students also have difficulty taking down notes; because they lose focus, the lecture is moving too fast or because they do not understand the material. If a student suffers from any of these issues, their notes may not be comprehensive, and could even be illegible, making learning the subject very difficult.

Some students may also have learning disabilities such as ADHD or dyslexia. Students with such conditions will find it a lot more difficult to concentrate, follow, or understand lecture material first time around.

Many want to depart from traditional learning methods such as lectures and move on to more 'modern day' teaching methods, like interactive learning applications. These applications enable students to gain a comprehensive understanding of a topic through hands-on learning; interacting with a topic, being fully immersed in it, physically seeing how the principle works, and being able to interact with it allows people to gain a fuller understanding of a topic than if they were simply told the information and expected to remember it.

In response to this problem, this project seeks to demonstrate how various principles of lighting/effects and physics - as used in digital media such as videogames and animation, can be used, demonstrated and taught to a user through the use of a software package; namely Unreal Engine 4.0. Planning to create an interactive piece of software that will be used to showcase various physics and lighting/effects principles used in modern video games, animation and other applications to the user. Demonstrating and teaching them how they are made to seem as lifelike as possible, and to allow the user to interact with objects in the environment to see what these principles are, how they are simulated and why they are needed.

### 3. Technical Research

#### 3.1. Game Engines

##### 3.1.1. Unity

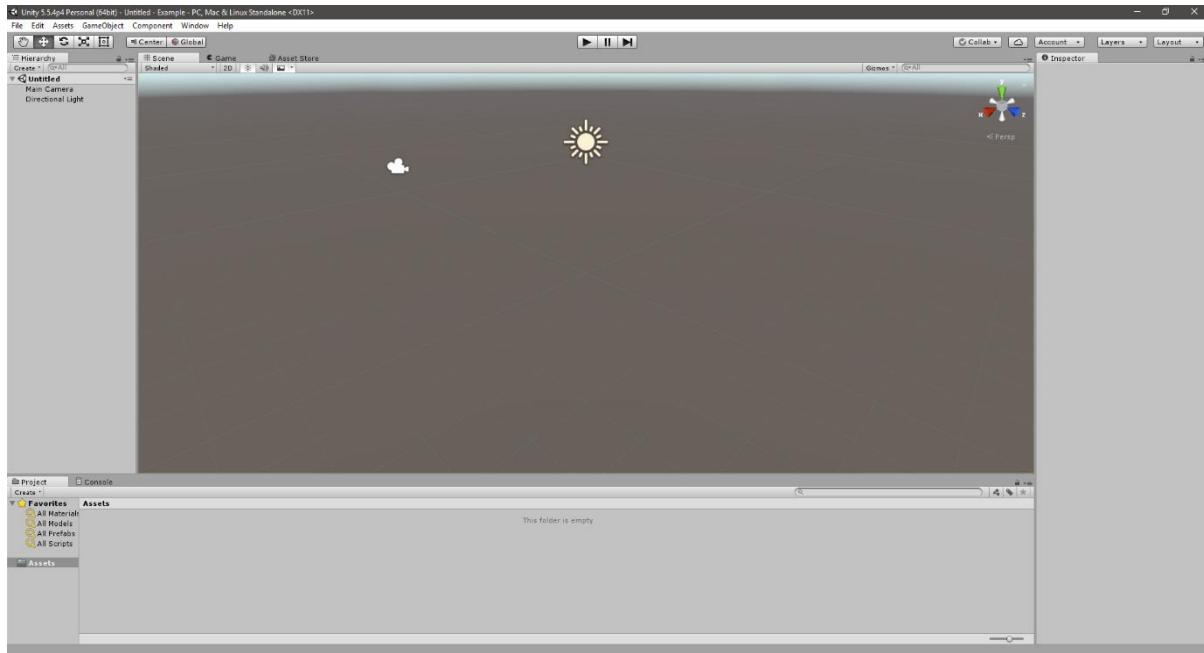


Figure 1 - Unity 3D Interface

Unity is a video game engine developed by Unity Technologies, currently running at version 5. Released in 2005 for Mac systems running OS X, it is now available on around 30 different platforms [2]. Unity supports 2D and 3D graphics and allows users to script objects using C# or JavaScript and offers drag and drop functionality for those less experienced in programming.

Unity supports over 30 different platforms, including PC, games consoles, mobile and the web. Unity makes use of the Direct3D API on Windows and Xbox One; OpenGL on Windows, MacOS and Linux; OpenGL ES on mobile platforms; and WebGL on the web.

###### 3.1.1.1. Pricing

Unity has four different pricing models; Personal, Plus, Pro and Enterprise.

The Personal Edition is for “beginners, students and hobbyists who want to explore and get started with Unity” [3]. This edition is free for commercial projects that have an annual gross revenue under \$100,000. It supports up to 20 concurrent multiplayer users.

The Plus Edition is for “creators who are serious about bringing their vision to life and plan to publish” [3]. This edition costs \$25 per month, and is for commercial projects that have an annual gross revenue under £200,000. It supports up to 50 concurrent multiplayer users. In addition to this, it also offers performance reporting services.

The Pro Edition is for “professionals who need complete flexibility and crave advanced customization” [4]. This edition costs \$125 per month, and has no limit on annual gross revenue. It supports up to 200 concurrent multiplayer users, and offers premium support in addition to all the features of the lower editions.

The Enterprise Edition is for “teams of 21 or more”, offering “custom solutions designed to support your organization’s creative goals” [5]. The cost of this edition is negotiated; there is no limit to

annual gross revenue and uses a custom multiplayer offering support for as many concurrent users as necessary. This edition also comes with access to the Unity source code.

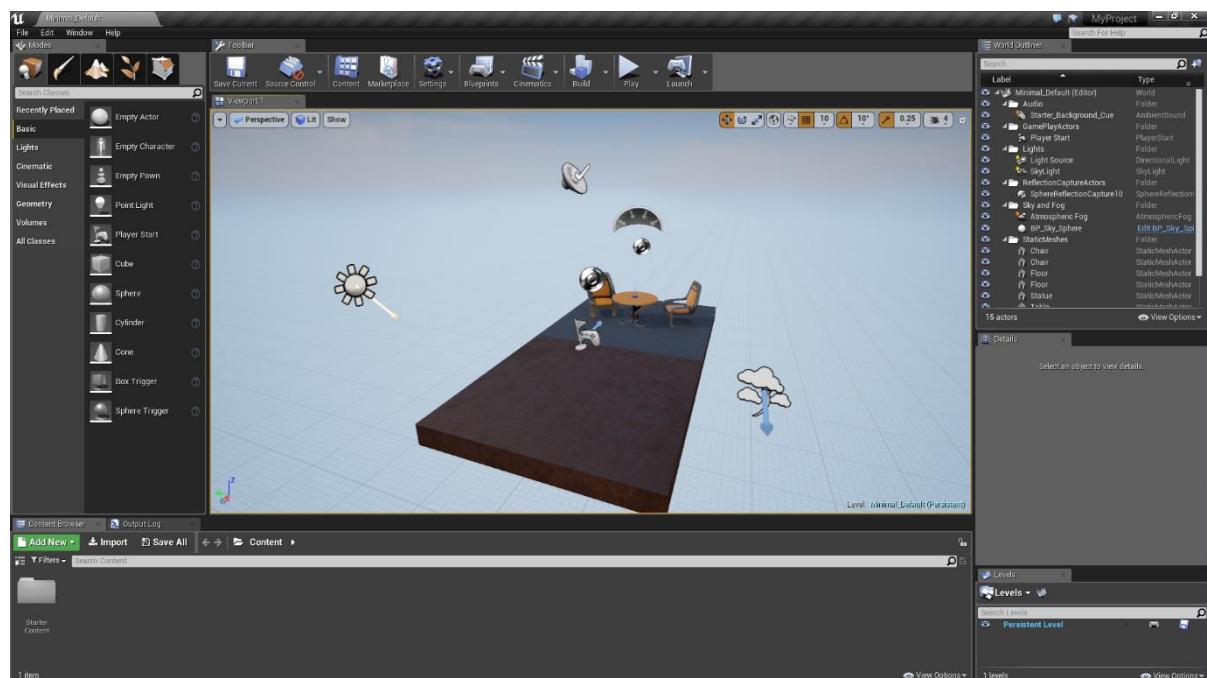
### *3.1.1.2. Advantages*

- Unity offers a vast selection of features, it is extremely flexible; can be used effectively to create many different types of games. It has native 2D capabilities, allowing for the creation of sprites, sprite sheets, 2D graphics and physics etc.
- Unity is very easy to use thanks to its simple, user friendly interface. This minimises the time spent getting used to the engine and how to use it, maximising time spent developing.
- As explained above, Unity has excellent cross-platform capabilities; allowing development for: Android, Windows Phone, iOS, PlayStation 3/4, Xbox360/One, Wii U, PC, Web browsers, PS-Vita, Linux, Windows, VR and more. This makes it almost effortless to port projects to other systems.
- Offers excellent support for 2D and 3D content (psd, png, jpg, gif, tiff, bmp, tga, dds etc.), also offers native support for 3D modelling applications such as 3DS Max, Maya, Cinema 4D and Blender. Also has support for common 3D files like: fbx, dae, dxf and obj.
- Unity's online asset store offers almost limitless free and paid content for use in the editor. This allows a solution to be made without any modelling, texturing or sound design, useful when time is a major factor in the development of a project.
- Unity is one of the most popular games engines for personal use, and as such has a very large online community. Any problems will most likely have been solved by someone online, making development much easier.

### *3.1.1.3. Disadvantages*

- Unity offers no real modelling, editing or building features; only offering the creation of primitive objects. This makes it difficult to white-box levels. All modelling must be done in third party software (3DS Max etc.), while there is no internal solution for modelling, Unity does offer very good support for modelling applications, in addition to the asset store (See advantages above).
- Unity has many additional costs, so it can be quite expensive for larger teams that need access to additional services, source code, or professional support.

## **3.1.2. Unreal**



*Figure 2 - Unreal Interface*

Unreal is a video game engine developed by Epic Games, currently at version 4. Initially used in 1998 for the first-person shooter video game ‘Unreal’, releasing for free to the public in 2009 as the Unreal Development Kit (Previously games could only be sold with a licence from Epic Games) [6]. Unreal Engine 3 became very popular with the Xbox 360 and PS3 generation, being used for many games including Gears of War, Bioshock, Borderlands and Mass Effect. While primarily used to develop first person shooter games, it has been utilised for many other genres.

#### *3.1.2.1. Pricing*

It used to be that Unreal could have a cost in the millions, but now it is free. The free version includes full access to the source code, but you must pay 5% of gross revenue after the first \$3,000 per product, per calendar quarter [7]. This is to eliminate the need to pay royalties by paying an upfront fee.

#### *3.1.2.2. Advantages*

- Unreal features a very strong, seasoned and robust level design tool set, with excellent white-boxing abilities and node based interfaces for shaders, scripting & animation.
- Features a built-in library of assets, meaning that modelling experience is not required to make a game. More assets can be downloaded from the Unreal Marketplace, including pre-built levels, models, sounds and textures.
- Very well documented, and has a strong online modding and development community, so there is a great deal of information available online.
- Similar to Unity, Unreal offers ‘blueprint visual scripting’, where objects can be scripted by blueprints rather than code, useful for those who lack programming knowledge.
- Great graphics; better than Unity, but not as good as CryEngine.
- Good cross-platform capabilities: PlayStation 3/4, Xbox360/One, Wii U, PC, Android, Windows Phone, iOS, web browsers.

#### *3.1.2.3. Disadvantages*

- Has different workflows than Unity, if coming from Unity development then it may take time to get used to the engine due to significant differences.
- Has quite a steep learning curve, especially coming from Unity. Meaning that some time may be required before one can develop at full capacity.
- Adding content is slightly more complicated, as with other disadvantages, this adds to the learning curve and may take time to get used to.
- As stated in the description above, it is primarily used to develop FPS games. Due to this, it is harder to develop other genres of game, but it has been done before.
- Could require a high-end computer to run on, depending on how detailed the game is.
- The marketplace is not on the same level as the Unity store, this could make finding content harder, and may even mean that some content simply isn’t available.

### 3.1.3. CryEngine

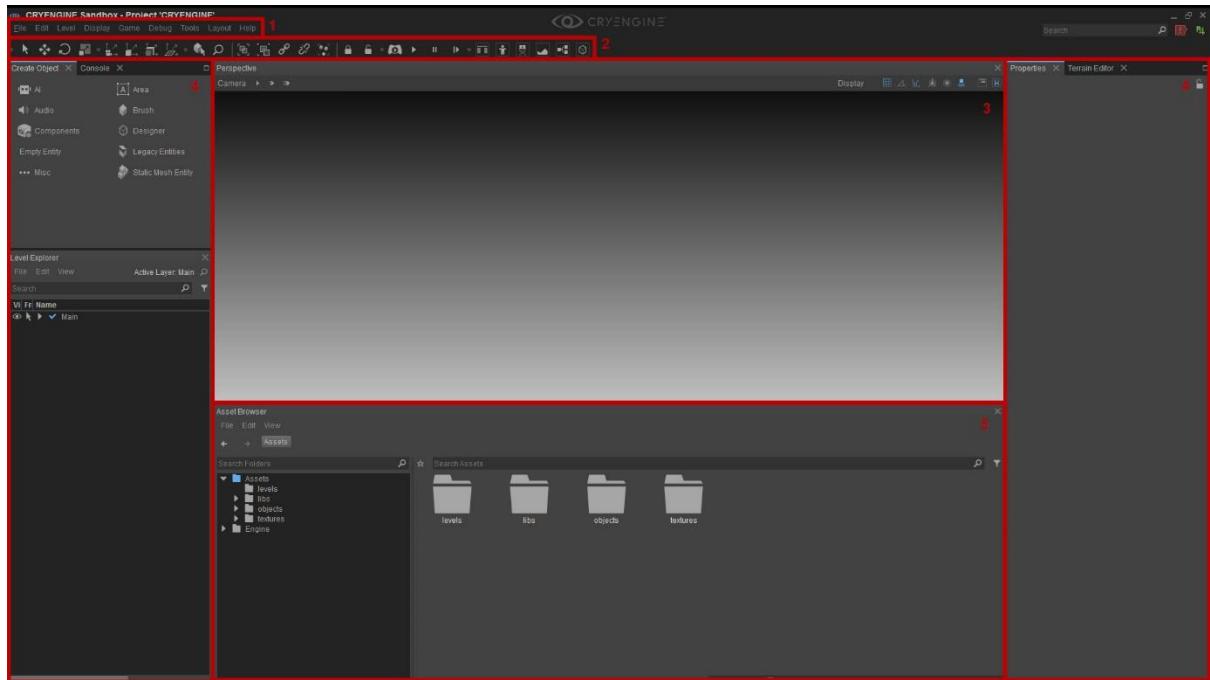


Figure 3 - CryEngine Interface

Image sourced from: <http://docs.cryengine.com/display/CEMANUAL/CRYENGINE+V+Interface>

CryEngine is a video game engine developed by Crytek, currently at version 5. CryEngine started as a tech demo for Nvidia in 2001, and was initially released in 2002, being used for the video game Far Cry. CryEngine is a sandbox editor focused on open world levels and gameplay, offering a real-time editor that allows the game to be played while editing.

#### 3.1.3.1. Pricing

CryEngine is free, you get the full engine source code, and access to all features. It operates on a pay what you want model, where you can pay what you want for additional support. Crytek offer memberships for either £45 or £130 per month for additional support and training. In addition to this, they also offer a full enterprise license for bespoke Crytek support. Crytek also has something called the Crytek Indie Development Fund, wherein teams can apply for funding on their indie game projects.

#### 3.1.3.2. Advantages

- A very powerful engine; usually looks to the future of technology and pushes the limits of what modern systems can do. CryEngine's graphics capabilities far surpass Unity and Unreal, possessing state of the art lighting, time of day simulation and atmospheric effects.
- Has powerful level design features, great for sandbox games. The real-time editor allows the game to be played in the editor, cutting down on time spent on building the game just to test something.
- Has a focus on procedural level design, allowing landscapes to be made procedurally rather than from scratch, greatly shortening development time.
- Very polished and familiar interface makes getting used to the software easier.
- Like Unreal and Unity, CryEngine possesses an alternative to scripting. The flow graph node based scripting tool allows objects to be scripted through flow charts similar to the blueprint system in Unreal.

- Has built-in modelling tools for sophisticated mesh and UV editing. Makes it effortless to create assets for levels from within the editor.

### *3.1.3.3. Disadvantages*

- Much harder to learn than the other engines, there is a steep learning curve and there is limited support so finding help will be harder.
- In addition to this, the online community is much smaller than Unreal and Unity and there are limited assets, examples and tutorials online.
- Not very stable compared to the other engines, saving often is imperative because the editor is very prone to crashing.
- Like Unreal, it is harder to create games other than first person shooters. Open world FPS games in tropical settings are the best fit for using CryEngine.

### *3.1.4. Chosen Engine: Unreal*

Unreal Engine was chosen for this project because it best suits the needs of the project scope. The solution requires an engine with strong graphical capabilities. Unity is, for lack of a better word, too simple to meet these requirements; its graphical capabilities are far surpassed by other engines and only simple primitive objects can be made in the editor, making the initial white-boxing of the level difficult.

CryEngine is on the opposite end of the spectrum, it is an extremely powerful engine with graphical capabilities far greater than Unity and Unreal; it has very strong level design features and a polished interface. But, it has a much steeper learning curve than the other two engines, with very limited support, a small community and good, official support only available to those with a membership. With this, and its known instability, CryEngine was believed to be too ‘overkill’ to suit the project’s needs.

Unreal Engine is the best fit for this project, sitting nicely in the middle of CryEngine and Unity. With its robust level design tool set, creating the level/s for the project will not pose any unnecessary challenges - Unlike Unity with its simple toolset. Unreal’s built-in library of assets and marketplace will make populating the level with high quality resources after the initial white-boxing stage easy without prior 3D modelling experience. The Unreal Marketplace is not on the same level as the Unity asset store, but is better than CryEngine’s store and its library of built in assets provides many resources to work with from the get go. Being a very well documented engine, there is lots of information and many tutorials online, any problem has likely been solved by someone in the community. However, CryEngine has a relatively small online community making it much harder to get information about any potential problems or solutions. Finally, Unreal possesses a great graphics engine, far surpassing Unity, which is essential for this project. CryEngine’s graphics capabilities are much better than Unreal, however, CryEngine comes with many disadvantages that would make choosing it over Unreal unwise.

## 3.2. Physics and Lighting/Effects Principles

This section details the research performed into the different principles that are to be implemented into the application. Most of this research was performed prior to development, but some of it was performed in tandem with the implementation process.

### 3.2.1. Collision Detection

Collision detection is the process of identifying if two or more objects have intersected. There are many types of collision detection, the one that will be demonstrated in this application is line to triangle collision or ray tracing. This involves using the linear equation  $ax + by + cz + d = 0$  which defines a plane [12].

$a, b, c$  = The planes normal. I.e. the vector that is perpendicular to the plane surface.

$d$  = the dot product of  $a, b, c$  and any point that is known to be on that plane. This can simply be one of the triangles vertices.

$x, y, z$  = The point being tested against the plane.

The result of this equations is the perpendicular distance from the point to the plane. This will be 0 if the point is on the plane, negative if the point is behind the plane and positive if the point is in front of the plane.

Assuming an object is constructed of triangles, a ray trace is performed on all triangles in that object. Once a collision is detected with any of the triangles it can be said the line has intersected with the object. The actual intersection point can also be calculated at this point with a few more calculations.

At a triangle level this can be broken down into 4 checks.

1. Does the line intersect with the triangles face plane? Calculate the intersection point if it does.
2. Is the intersection point on the positive side of edge plane 1
3. Is the intersection point on the positive side of edge plane 2
4. Is the intersection point on the positive side of edge plane 3

If all 4 of these checks are true then the line has intersected with the triangle and we also have the intersection position. As soon as any of them are false, we can bail out and move on to the next triangle.

There are some areas of this process that can be pre-processed such as the calculation of the face and edge planes, though these will most likely be calculated in objects local space if the object is dynamic (needs to move about in the world). In this case when the collision check is performed, the line segment points will be transform into the objects local space before performing the collision checks. This avoids the expense of calculating all of the planes at collision time.

### 3.2.2. Cloth Simulation

Cloth simulation is the term used when simulating real cloth in 3D computer graphics. The three ways that this can achieved are geometric, physical, or particle/energy simulation. The one demonstrated in this application is particle/energy.

The basic approach is to use particles spread across the surface of the cloth with an even distribution. The particles are constrained to each other using linear constraints. These maintain the distance between one particle and the next. Stretching can be achieved by allowing the constraint to move above or below their original length.

Some particles can be locked to a static position allowing the cloth to hang from fixed objects etc.

The particles can have physics applied to them using any form of integration. This application uses Verlet integration. Forces from wind, gravity and from collisions can be applied to the particles.

Once the physics is simulated and any collisions are resolved the constraints are applied to maintain the distance between the particles.

### 3.2.3. Flocking Simulation

Flocking is the behaviour exhibited by groups of animals such as birds, fish (shoaling) and insects (swarming). Flocking simulation was first developed in 1987 by Craig Reynolds with the program Boids [13].

In computer simulations this behaviour can be modelled using three basic rules that affect the direction of each animal (bird).

1. **Grouping:** Turns a bird towards the centre of its group.
2. **Alignment:** Turns a bird to line up with the average direction of its group.
3. **Separation:** Turns a bird away from its immediate neighbours to maintain a fixed spacing distance.

### 3.2.4. Types of Lights, shadows and Lighting Effects

Lighting in computer graphics is an approximation of a real world phenomenon when light is bounced off a surface and is received by the eye.

This is approximated (modelled) using Lambert's Cosine Law. This calculates the amount of reflected light from a surface using the cosine of the angle between the surface normal and direction of the incoming light.

Various types of lights can be modelled including Directional, Point and Spot lights.

Directional Lights: Produce light from an infinite source. This produces a parallel light source and is generally used to simulate sunlight.

Point Lights: Produce light from a single point in all directions. These can be likened to a light bulb.

Spot Lights: Produce light from a single point in a direction. These can be likened to a car headlight.

Another part of the light model is distance attenuation. This is the effect of light diminishing the further it gets from the source. This can be a linear calculation just using the distance from the light or can use the more realistic exponential calculations meaning the light energy reduces quicker, the further it gets from the light source.

Shadows are cast when the light from a source is occluded. In computer graphics this is normally achieved using a shadow map. This takes the form of a depth buffer rendered from the point of view of the light. Objects rendered in the scene use this shadow map by calculating each of its pixels distances from the light. Then the shadow map is read at the location that maps to the light's direction to get the scene distance from the light at that point. These 2 values are compared and if the scene distance is less than the object's distance from the light it is in shadow and this light does not contribute to the final pixel colour.

### 3.2.5. Post Processing Effects

Post processing is part of the rendering pipeline. It takes the final scene output buffers and performs various image processing techniques on the pixels to produce the final output.

Effect such as Anti-Aliasing, Automatic Exposure (Eye Adaption), Depth of Field, Lens Flare Vignette and Bloom can be achieved.

### 3.2.6. Other Principles

The following principles were not included in the final application, the resources used for research are still listed in the references section, but nothing will be written about these principles here.

- Molecular Simulation (Water/Sand)
- Materials and Material Properties
- Reflections

## 4. Requirements and Analysis

This Software Requirements Specification (SRS) details the requirements to build a piece of interactive learning software, with the purpose of demonstrating physics and lighting/effects principles to users in the Unreal 4.0 Engine, specifically version 4.14.3.

This Software Requirements Specification was written in accordance to the standards for software requirements specifications as detailed in the “BS ISO/IEC/IEEE 29148:2011 Systems and Software Engineering - Life Cycle Processes - Requirements Engineering” document, available from the IEEE Xplore Digital Library.

### 4.1. Purpose

The purpose of this application is to demonstrate various principles used in video games to simulate physics and lighting/effects. To allow the user to interact with objects in the environment to show what these principles are, how they are simulated and why they are needed. This software could be used to demonstrate the capability of the engine; showing what one can do with it to make a realistic game or simulation. Additionally, showing aspiring games development students considering the field what things go in to a game or simulation to make it look as life-like and realistic as possible, and what kinds of things they will be studying at university.

### 4.2. Scope

The interactive learning software, hereon to be referred to as ‘the application’, will be accessible by anyone without need for registration, hereon to be referred to as ‘the player’. The application will start by displaying an introductory message to the player, stating what the application is for, and how to use it. Once this message has been displayed, the player will be free to interact with the features of the application.

After the introductory message, the player will find themselves in the ‘starting zone’. This is the area in which the player starts. The starting zone acts as a hub for accessing the different areas of the application; of which there will be three: The ‘Physics Area’, the ‘Lighting and Effects Area’, and the ‘Sandbox Area’. The starting zone will consist of a circular paved area with a fountain in the middle. There will be three paths splitting off from the starting zone – one for each area of the application, and these paths will be labelled to show the player what they lead to. Each area will contain various demonstrations that will be described below; these will be explained in further detail in the design section.

The physics area will consist of various demonstrations. Each of which will describe a researched physics principle, show the player how it is implemented, and show them what the finished implementation looks like.

The first of these demonstrations will be of collision mechanics. The demonstration will be housed in a display area in the middle of a small building. Surrounding the demonstration will be various objects the player can interact with after viewing the demonstration to test what they have learnt from it. The player will be equipped with a weapon that shoots projectiles that stick to these objects once colliding with them.

The next demonstration will be of cloth simulation. As with the last demonstration, this one will be housed in a small building, but rather than a single animated display in the centre of the room, it will be a series of small demonstrations lined up next to each other, taking up one full side of the room. These demonstrations will show to the player the different stages of simulating a cloth (creating the vertices, constraining them etc.). There will also be a display in the middle of the room; a cloth will be draped over various shapes that will move around underneath it, showing how good the cloth simulation is. The room will then be filled with other cloths and ropes (Made the same way as cloths

but with only 1 axis); with objects close by them, the cloths and ropes will then be blown into these objects by the wind, further demonstrating the effectiveness of the cloth simulation.

The physics area will also have a demonstration area for flocking mechanics. This demonstration will showcase simulated flocking mechanics to the player. This demonstration will not be contained within a building, but will have a designated area outside. The player will walk onto a marker to activate the demonstration, after this a number of birds will spawn and start flying around randomly, flocking together like real birds. The player will also be equipped with a weapon that fires two types of projectile; one the birds avoid like an obstacle, and one they follow as if it were another bird. This demonstration will be accompanied by a textual description of the flocking mechanics, and what rules are needed to implement it effectively.

The final demonstration in the physics area will be of molecular simulation. This demonstration will simulate water and/or sand to the player using molecules. It will take place, again, in a small building. The walls of which will be lined with small cabinets, not dissimilar to arcade machines, that will contain various moving parts and obstacles. These cabinets will fill with the simulated sand or water when the player moves near them; pouring in from the top, the molecules will clump together and move around as if they were a single body of water or sand. The middle of the room will contain a small, animated display, similar to the collision room, explaining the various rules that are applied to the molecules.

Lighting and effects is far too broad a spectrum to cover in detail here; a whole application could be created just focusing on lighting techniques alone. Considering the time frame for the project, it would be unrealistic to implement many of the lighting principles available in Unreal. Therefore, this lighting and effects area will only showcase the basics of lighting and effects, and show the play why they are important.

The lighting and effects area will consist of various demonstrations, each of which will describe a researched lighting/effects principle, show the player how it is implemented, and show them what the finished implementation looks like.

The first of these demonstrations will be of types of lights, shadows and lighting effects. The demonstration will take place inside a small building as others have. The windows will all be blacked out so outside light does not have an effect on the demonstration. There are four types of light in Unreal: Directional, point, spot and sky. One of each of these lights will be placed in the room along with various objects, and the lights will be toggled, cycled through etc. to show what each does, and what effect it can achieve. As with other demonstrations, this will be accompanied by a textual description. Shadows will also be described, and these toggled on and off, to show the importance of shadows, and how much of an effect they have on a scene. If this demonstration feels like it is lacking compared to others, a smaller demonstration on lightmass global illumination will also take place in an upstairs room in the same building. This demonstration will showcase global illumination, and show players what it is, and what effect it has on a scene.

The next demonstration will be of post processing effects. There are a multitude of post processing effects available in Unreal; only six of them will be covered in this demonstration: Anti-aliasing, eye adaption, depth of field, lens flare, vignette and bloom. This demonstration will be simple; the post processing effects will simply be toggled one by one, and explained through textual description what each one does. An effect will be turned on, and text will appear describing it, the effect will then turn off and on to show a comparison between no effects vs. effect.

The lighting area will also have a demonstration for materials and material properties. This demonstration will have a variety of objects, textured with different materials. These materials will have different properties, such as roughness and metallicness. Each material will also have various

texture maps, such as normal maps. These give the texture more depth; control how light interacts with the material, and gives the illusion of greater detail than is possible.

Each material will be displayed on an object to the player, each will also be accompanied by a display of each texture map the object has. So if a material has a texture, normal map and diffuse map. The material will be displayed on one object, which will be accompanied by three other objects showing just the texture, diffuse, and normal maps that make up the material. These materials will have textual displays describing what each map does, and how it is used to allude to more detailed materials. The material properties will also be described here, each material changing their properties to show how they affect the final outcome. If this demonstration feels like it is lacking compared to others, a smaller demonstration on post process materials will also be included. This will simply display more objects, each with a separate post process material, accompanied with textual descriptions of each.

The final demonstration in the lighting area will be of reflections. This demonstration area will be different to others; within the demonstration building will be a small scene containing lights, mirrors, reflective surfaces etc. This demonstration will show how reflections are important to a scene, and demonstrate things like screen space reflections. To demonstrate this, reflections will be toggled off and on to show the difference in quality of the scene. The demonstration will also be accompanied by a textual description of reflections and why they are important while the player is walking through the scene.

The sandbox area will be a mixture of both the physics and lighting/effects area. It will be an informal blend of elements from the two areas, there will be little to no explanations or descriptions about the demonstrations. This area will be more of a playground for players to have fun with the different elements of the application, such as the different weapons, the cloth, or the sand. To ensure the informal style is kept, this area will not have a set design, and will evolve naturally depending on what is added to the application. This area will be made at the end of development, and is not an important aspect; therefore, it can be cut from the application without affecting the core objectives of the software.

### 4.3. Product Perspective

#### 4.3.1. System Interfaces

The application will not require any additional interfaces to any other systems; it will be a standalone application, with all functionality being self-contained.

#### 4.3.2. User Interfaces

The application will have a basic user interface, consisting of only a few minor aspects. Both the user interface and the heads up display of the application will be described in further detail in section 5.4 of this report.

The user interface of the application will be simple, consisting of only a small crosshair to convey to the player what direction the projectile from their equipped weapon will fire. It will also include either a textual or graphical representation of which projectile their weapon is set to fire. (The projectile changes based on what demonstration the player is visiting).

The heads up display of the application will simply be a text box at the bottom of the screen for showing descriptions of each demonstration. This will only appear when the player is viewing a demonstration that has a textual description behind it. The text in this box will be large enough to be easily readable by anyone, and should stay on screen long enough to be easily readable.

The player will physically interact with the application through the use of a keyboard and mouse. They will use these peripherals to control the in-game character, and interact with demonstrations.

The player will use the mouse to control the camera, the W, A, S and D keys on the keyboard to move, and the space bar to jump.

Any error messages displayed should be shown on-screen in a short format, and for a more detailed view, the full error should be printed to the console.

#### 4.3.3. Hardware Interfaces

The application will not require any addition hardware interfaces, apart from those required by Unreal itself; Graphics card for rendering, processor for calculations, secondary storage for install for example. All of these interfaces however, will be controlled by the engine itself, and not available for the application or its creator to modify.

#### 4.3.4. Software Interfaces

The application will not require any additional software interfaces to function. The software will be completely standalone, and will not derive any functionality from other software. The host machine may however, require packages such as DirectX and the Visual C++ Redistributable to be installed, as they are required for 3d rendering.

The first version of the application will only be compatible with Windows machines. This first version will be the version created for this project. Later versions could be expected to be compatible with many different platforms. Using the Unreal Engine, it is very simple to port applications to any of the supported systems. System-specific changes have to be made (touch screen controls etc.), and the performance must be kept consistent on the new systems hardware, but other than that, the application just needs to be packaged in a different format.

#### 4.3.5. Communications Interfaces

There will be no additional communications interfaces to the application, it will be completely offline, and does not have the need to communicate or be communicated with.

#### 4.3.6. Memory

The application should take up as little amount of secondary memory as possible when installed on to a system. The application should ideally take up less than 5GB of secondary storage space, but anything below 10GB is acceptable.

The application should be well optimised, and only use the resources that it needs for operation. As such, the application should not take up a great amount of primary memory, and should be able to run efficiently on systems with 4-8GB of primary memory installed.

#### 4.3.7. Operations

The player of the application will be in control of a moveable character in the level. This character will be controlled by the movement keys, and the character's view by the mouse. This player will interact with various objects throughout the application via the character; this operation is controlled by the user of the application (the player).

The application will have many demonstrations, most of which will be automatically sequenced operations, carried out by the application, looped for the player to watch over and over. Other parts of these demonstrations may not be automatically sequenced, but will still be simulated and controlled by the application. For example, the simulated cloth blowing in the wind; this is not automatic, it is a dynamic simulation, but it is still controlled by the system.

The application will not handle any user data, so there are no security concerns. Furthermore, no outside data will be handled whatsoever, so there will be no provisions for data processing, or data backup.

#### 4.3.8. Site Adaption Requirements

As the application is only intended for use on Windows machines, there are no site adaption requirements as such, since the application is only planned to be used on a single installation.

The application will however need to be installed on a Windows machine, with the DirectX and Visual C++ Redistributables installed in order to function properly.

As mentioned previously, in the future, the application could be planned to be ported to other systems. The porting of the application to any system that supports Unreal is made very easy thanks to the Unreal Editor.

### 4.4. Product Functions

The application will feature many demonstrations of physics and lighting/effects principles in the two respective areas of the level. Each demonstration will take place in its own sub-area of either the physics or lighting/effects area. These sub-areas will either be a small building made up of a single large room, or a small open space depending on the demonstration. Each sub-area will have a main demonstration of a principle, and some form of interactive aspect, if applicable. At the very least, a demonstration should have an automated demonstration describing a principle. Each of these demonstrations should be accompanied by a textual description, which should be used to describe what is happening in the demonstration, and to provide more information about why it is used.

The demonstrations in the application should be:

- A collision detection demonstration, showcasing how collision is detected and calculated between two objects.
- A cloth simulation demonstration, showcasing how cloth is simulated, and how it interacts with forces such as wind.
- A flocking mechanics demonstration, showcasing how flocking birds are simulated, and the simple rules that they follow.
- A molecular simulation demonstration, showcasing how sand or water can be simulated using molecules that clump together realistically.
- A simulation demonstrating different types of lights, and how they are used.
- A post processing effects demonstration, showcasing post processing effects and what effects they have on a scene.
- A materials and material properties demonstration, showcasing how materials are made up, and how the properties and different maps used in them helps to create more detail.
- A reflections demonstration, showcasing how big of an impact reflections has on the quality and realism of a scene.
- A sandbox with a mixture of elements from all the above demonstrations, presented in an informal manner.

### 4.5. User Characteristics

The target audience for this application is split into two distinct areas; people currently enrolled on a university course, and people thinking about enrolling onto a course.

The first type of user for this application is the university student, who is currently enrolled onto a computer science, computer games design, or other computing related degree. This user could be anywhere between the age of 18-35, male or female, and are expected to have at least some basic prior knowledge and/or experience with computing concepts from their degree, and a basic proficiency in mathematics.

The second type of user for this application is the young person thinking about joining a computing related university degree. As with the first user type, they could be any age up to 35 years old, but they are expected to be much younger, typically around 16-18 years old. The user is again expected to be either male or female, and should have an interest in computing and computing related topics. Again, a proficiency in maths may be required to understand some aspects of the demonstrations.

While this application is not meant for those with learning disabilities such as ADHD or dyslexia, it could certainly set a new precedent for digital learning. Those with such learning disabilities could find it much easier to concentrate and understand the content proposed in the application, versus if it were delivered to them in a traditional lecture. The application also lets them revisit anything as many times as they need to if they don't understand it.

#### 4.6. Limitations

There are not many limitations to this application; there are only two main limiting factors when it comes to using this application. The system the application is installed on should meet the minimum system requirements for running Unreal Engine, otherwise the application may not run at its intended performance, and could suffer negatively from this. The other main limiting factor is the user. Depending on the disability, the application could be very difficult for someone with disabilities to use. If someone does not have the full use of their hands, then controlling the character, navigating the level, and interacting with the demonstrations could be very challenging. Additionally, if the user does not have at least some basic prior knowledge, and isn't very proficient in mathematics, they could find some demonstrations confusing or hard to understand.

#### 4.7. Assumptions and Dependencies

Completion of the project is dependent on the following factors:

- Access to a computer system with capable hardware and the Unreal Engine Editor installed
- Access to 3D modelling software if any additional assets are needed to be created
- Proficient knowledge of the C++ programming language and Unreal APIs

For the success of the project, the following factors are assumed to be true:

- All the required assets will be available through the default content provided with Unreal, and through content provided on the asset store
- The researched principles are possible to implement in Unreal Engine 4.0
- Up-to-date and relevant tutorials and guides on how to implement certain things in Unreal are available
- The researched principles will all be able to be implemented within the time frame of the project.

#### 4.8. Apportioning of Requirements

The first version of the application (the version planned for the end of this project) will only be available in English, later versions of the application are expected to have multilingual support and narration for the demonstrations. But this is outside the scope of this version of the application.

## 4.9. Specific Requirements

At a high level, here are the minimum specific requirements for the application:

- The application should have a controllable character that the player can manipulate and use to travel around the level and interact with the demonstrations.
- The application should have three distinct areas, physics, lighting/effects, and sandbox.
- The physics area should contain four sub-areas containing demonstrations for the four researched physics principles.
- The lighting/effects area should contain four sub-areas containing demonstrations for the four researched lighting/effects principles.
- The sandbox area should contain a mixture of the elements from the previous two areas, presented in an informal manner, meant to entice people into using the application at fresher's fairs and other events.
- Each demonstration should be kept separate from others, i.e. in their own rooms/buildings, but all relating demos i.e. all in the same category, should be segregated together.
- At the very least, each demonstration should provide a description of the demonstrated principle and show player how it works. Ideally there should also be an interactive element accompanying the demonstration to further help the player grasp the concept.
- The playable level in the application should be easily expandable for adding more demonstrations/areas in the future.

## 4.10. Usability Requirements

The application should be simple and easy to use by anyone, no matter their age, or technical background. While some content within the application may require some prior technical or mathematical knowledge, the application itself should be usable by anyone.

The demonstrations provided in the application should be easy to follow and understand. They should however, not be oversimplified.

The application should effectively and efficiently teach people the researched physics, lighting and effects principles through the user of its demonstrations and interactive portions.

## 4.11. Performance Requirements

Since there is no inputted or outputted data to the system, the only performance requirements are with the application itself, so as such: The application should never drop below 30 frames per second on a modern, mid-tier computer system.

## 4.12. Design constraints

Unreal Engine has a list of standards put into place for coding, naming files and developing content. These standards must be followed when designing and implementing the application.

## 4.13. Standards compliance

The application will conform to all naming conventions and coding standards and guidelines of Unreal Engine.

Unreal Engine has coding standards and conventions put into place for a variety of reasons, which must be adhered to when coding in Unreal [9], [10], [11].

Unreal also has a list of external content development standards that any content intended for distribution must follow. While this project and the content contained within is not intended for distribution, these standards should still be followed just in case it is to be distributed in the future.

In addition to this there is also a list of general guidelines for content creation and level design that should be followed in order to improve performance. As performance will be a major factor in this project, these guidelines should be closely followed.

#### 4.14. Software System Attributes

##### 4.14.1. Reliability

The application should work reliably, it should not crash and not suffer from any slowdowns. To achieve this the application should be tested rigorously throughout development, and again at the end of implementation. All operations and code within the application should also be tested and optimised to ensure maximum performance is achieved.

##### 4.14.2. Availability

The application should not save any data, or progress of the players. It should reset completely when it is next used. Since the application is offline, and only available locally, it has no availability constraints or attributes.

##### 4.14.3. Security

Since the application handles no user data, does not require signing up, and lets anyone use it, there are no security concerns. The application should still however prevent people from modifying the application, this however, is handled by the Unreal Engine, when applications are packed as an executable there is no access to the source code, just a simple exe and package files.

The application will also not be available for public use at home, only in teaching environments and fairs, so no one will be able to gain access to the application.

##### 4.14.4. Maintainability

The application may be maintained or improved upon in the future by someone other than the original creator, in such a case these people will have never seen the code or project before. In addition to this, the original creator may go back to the project after a long time, and if it is not documented they may find it difficult to understand what they themselves did.

So to ensure easy maintainability, all sections of code should be thoroughly documented, commented and explained. All coding standards and conventions should also be followed to further improve ease of maintainability in the future.

##### 4.14.5. Portability

Being made in Unreal Engine, the application is extremely easy to port over to any of the various supported systems of unreal. The project simply needs to incorporate any system specific features (such as touch screen controls), make sure the application still meets performance requirements on the new hardware, and it can be packaged in the new format supported by the other system.

#### 4.15. Verification

The application will be tested using an agile approach throughout the development process; each element of the application will be tested while it is being implemented, and any errors corrected before moving on to the next element. The whole application will also be further tested once it is complete, to check for any errors that may have avoided detection in the implementation stage.

Furthermore, once completed the application will be tested by a variety of people and judged for its effectiveness. This will determine how effective the application is as a digital learning tool to teach people various concepts used in videogames, simulations, and other multimedia applications.

## 5. Design

### 5.1. Overview

#### 5.1.1 Purpose

The purpose of this application is to demonstrate various principles used in video games to simulate physics and lighting/effects. To allow the user to interact with objects in the environment to show what these principles are, how they are simulated and why they are needed. This software could be used to demonstrate the capability of the engine; showing what one can do with it to make a realistic game or simulation. Additionally, showing aspiring games development students considering the field what things go in to a game or simulation to make it look as life-like and realistic as possible, and what kinds of things they will be studying at university.

#### 5.1.2 Background

The application will start in a starting zone, offering the player information about what the application is and how to use it. They will be then free to explore the level. The level itself will be a single open region, with three different areas for demonstrating physics and lighting/effects principles, and a more informal sandbox environment for entertaining people and getting them interested in the application.

### 5.2. Features

#### 5.2.1 General Features

The application will demonstrate various physics and lighting/effects principles to the player, and show them how each works, and how it is implemented. Below is a list of the planned principles for implementation in to the application. All the below principles will be demonstrated to the player in one way or another, those applicable will be interactive to the player (E.g. collision detection may include an area where the player can interact with various physics objects). All demonstrations will be accompanied by a textual description of what the principle is, how it is simulated and why it is needed.

Physics Principles	Lighting and Effects Principles
<ul style="list-style-type: none"><li>• Collision Detection</li><li>• Cloth Simulation</li><li>• Fluid Dynamics/Molecular Simulation</li><li>• Flocking Mechanics</li></ul>	<ul style="list-style-type: none"><li>• Types of lights/shadows &amp; lighting effects</li><li>• Post processing effects</li><li>• Materials</li><li>• Reflections</li></ul>

Table 2 – Planned Features

#### 5.2.2 Gameplay

The player will control a character that they can move around the level, exploring different demonstrations, each showcasing different principles. Some of the demonstrations will have nothing to interact with, just a display accompanied with appropriate descriptions. For example, an area demonstrating different types of lights will have various lights and other objects but will otherwise not be interactive. The player will be able to walk up to the lights, and maybe manipulate some objects in the scenery, but it will remain mainly static. Whereas other areas will be greatly interactive; areas showcasing flocking birds for example.

There will be no endgame, goals or objectives. The player will just be given free rein to roam around the level exploring each demonstration. There will be only one large level containing many demonstrations, rather than an individual level for each. This is because each level would be very

small, and the player would only be there for a relatively short amount of time, meaning they would have to wait for the application to load each time they wanted to see another demonstration.

### 5.3. Game World

#### 5.3.1 Overview

The game world will be a large open map, split up in to three distinct areas, each of which will contain various demonstrations for each of the previously mentioned principles - These sections will essentially be self-contained miniature levels.

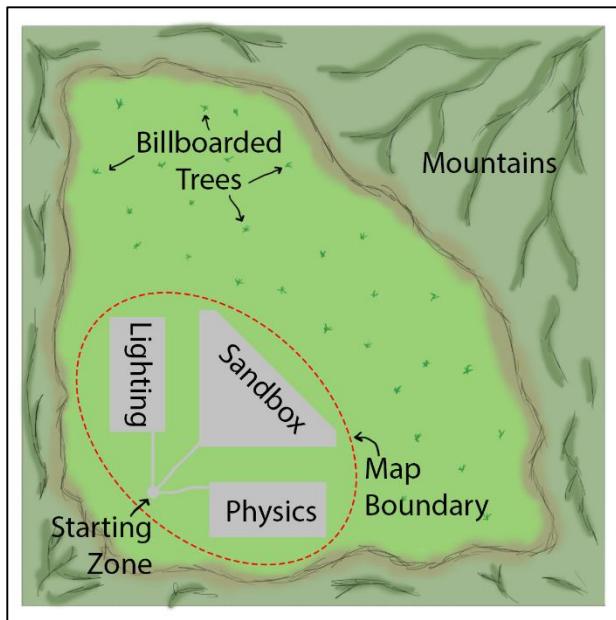


Figure 4 - World Overview

Figure 4 above shows an overview of the intended design for the level. The borders of the map are surrounded by mountains and hills, while this serves as a boundary for the level so one cannot see off the edge of the map, it will also be very visually pleasing, and will create a natural looking landscape.

The playable area of the map has a starting zone - where the player will start, a physics area – containing all the physics demonstrations, a lighting and effects area – containing all lighting and effects demonstrations, and a sandbox area – containing a mixture of elements from both areas. The playable area is marked by the red dashed line in the figure above, this line represents the boundary of the playable area, and in the implemented level it will be a blocking volume to prevent players from leaving. The surrounding area will be filled with billboarded trees (a 2D image of a tree that rotates to always face the camera). The trees will be far away, so a 2D image is all that is necessary, and at distance will give the impression of 3D trees.

#### 5.3.2 Camera

The camera will use a first-person view, this will ensure the player is not distracted by a player model taking up a portion of the screen when they are trying to see a demonstration in the level.

#### 5.3.3 Game Engine

The game engine that will be used is the Unreal Engine. Explanations have been given for the choice in engine in section 3.1.4 of this document. In short, this engine offers the best balance between graphical capability and simplicity and efficiency in level design.

### 5.3.4 Environment

The application will be set in a grassland, surrounded by hills and mountains to hide the borders of the map. The starting zone will be a simple paved area centred with a stone fountain. The three areas of the application will be urban environments with tarmac/asphalt, pavements and brick buildings housing the demonstrations.

## 5.4. User Interface

### 5.4.1 Overview

The application will include a simple heads up display. The main aspect of the HUD will be a textual display for displaying important information to the player about each principle when the player is viewing a demonstration. The application will also feature a simple crosshair so the player will know where the projectiles from their weapon/s will go. There will also be some form of either textual or graphical representation of what projectile the weapon is set to fire.

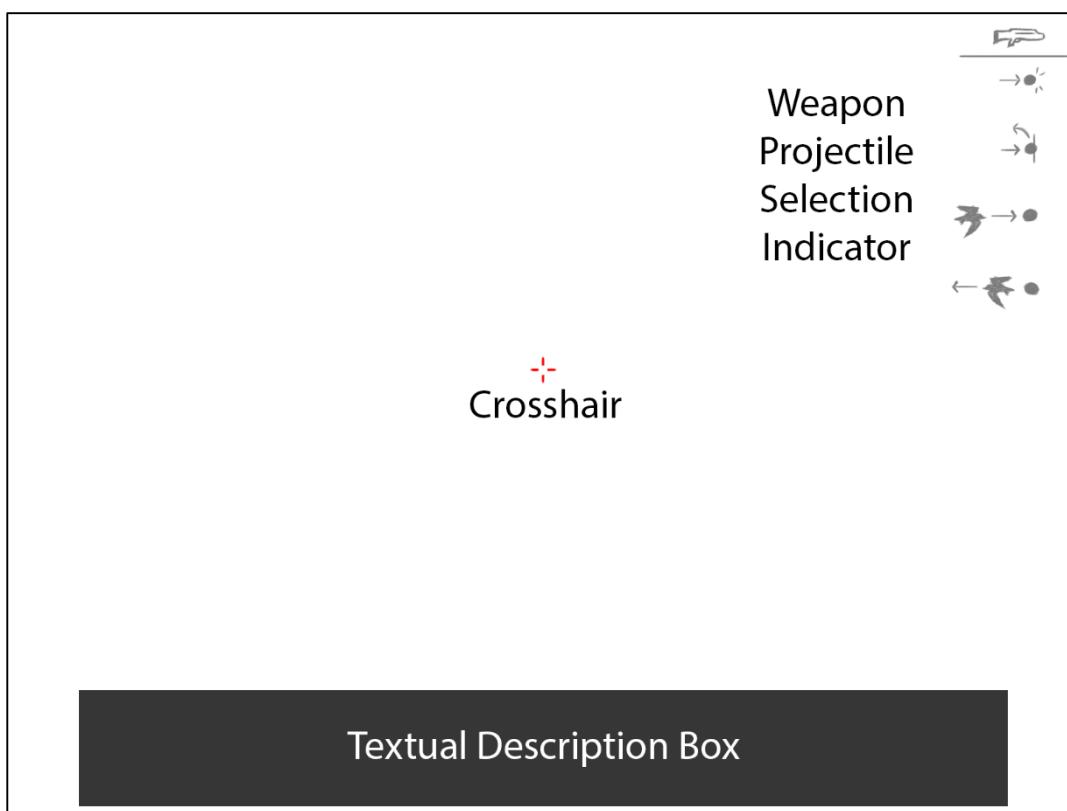


Figure 5 - User Interface

Figure 5 above shows the proposed user interface for the application. It consists of a crosshair, textual description box and a projectile selection indicator. The crosshair will show the player where the weapon will fire a projectile, and the weapon projectile selection indicator will show the player what projectile the weapon is currently set to fire. Some demonstrations will use special projectiles, such as the collision demonstration using sticky projectiles to demonstrate hit detection.

The textual description box will only appear when the player is viewing a demonstration. It will appear on screen, and contain descriptions and walkthroughs of the demonstration that is currently playing. The box will be able to cycle through multiple lines of text, so it is not limited to only a few words. It essentially acts as a subtitle box.

## 5.5. Level Designs

The following section details the designs for each area of the level, and each demonstration room within each area.

### 5.5.1. Starting Zone

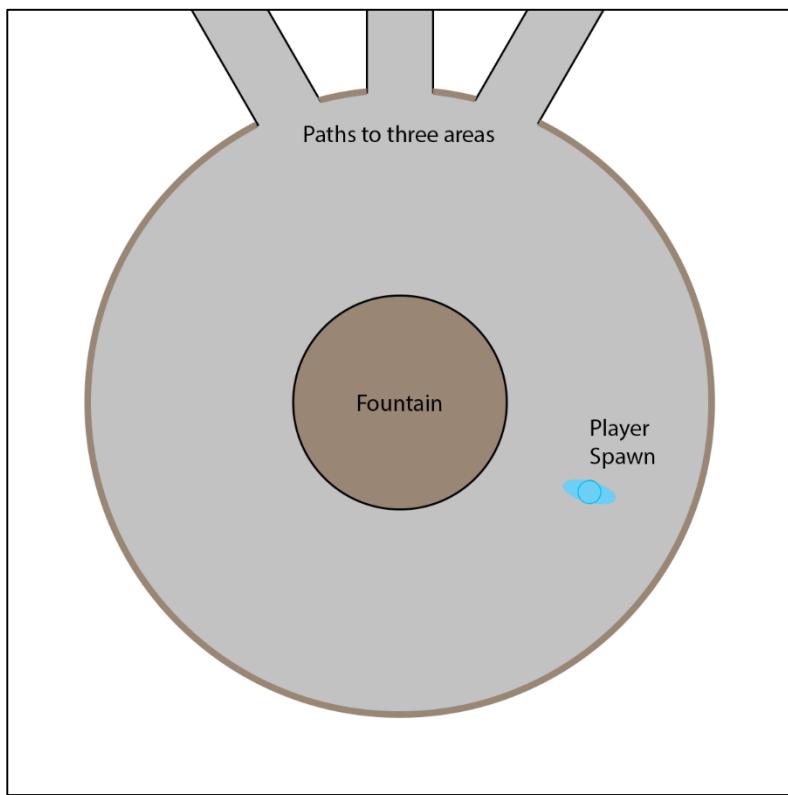


Figure 6 - Starting Zone

Figure 6 above shows the starting zone for the application. The blue shape marks the player spawn location, this is where the player will appear when the application is first opened. There will be a fountain in the middle for decoration, and to the top of the area will be three pathways to the various areas of the level.

### 5.5.2. Physics Area

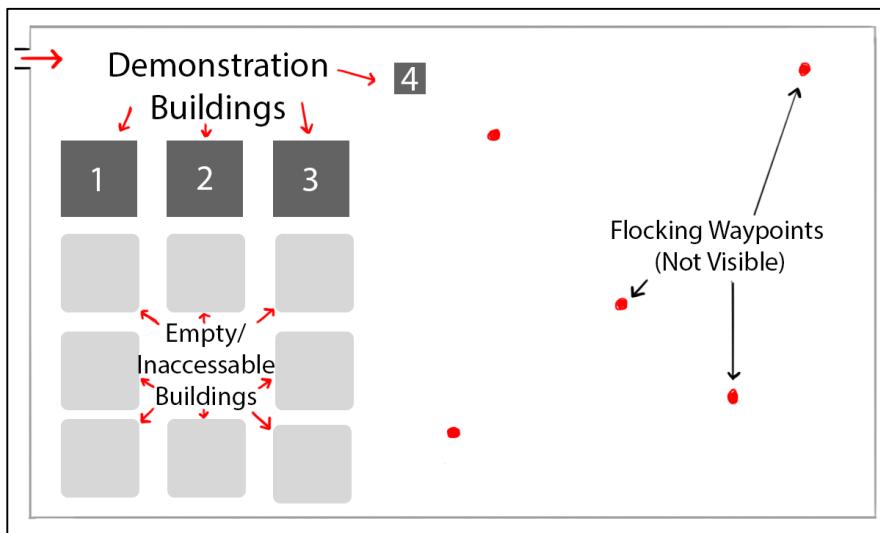


Figure 7 - Physics Area Overview

Figure 7 above shows the layout for the physics area - labelled in figure 4 as 'Physics'. The physics area will have four demonstration areas, these will be three buildings, and one outdoor display. There will also be a variety of inaccessible buildings acting as scenery. The entrance to the area is marked on the design as a red arrow in the top left. All demonstrations in the physics area will be accompanied by textual displays, explaining everything that is happening in the demonstration.

#### 5.5.2.1. Collision Demonstration

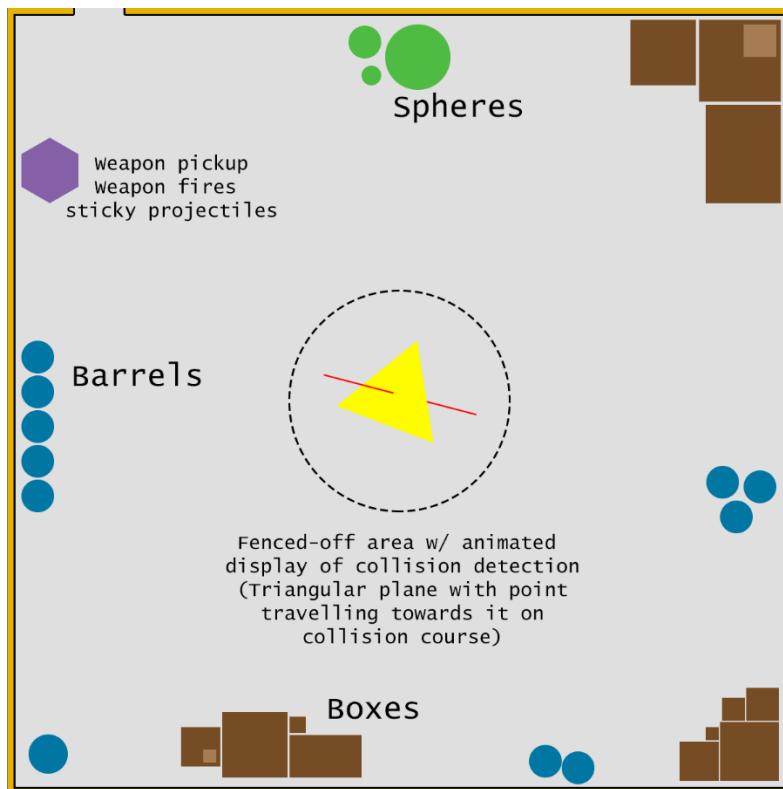


Figure 8 - Collision Building

Figure 8 above show the first of the demonstrations in the physics area. This is the collision demonstration area – the building marked '1' in figure 7.

The demonstration itself is the yellow triangle in the middle of the room, this demonstration is described in more detail in figure 9 below. The rest of the room will be filled with various objects, each of a differing size, shape and weight. The player will be equipped with a weapon that fires sticky projectiles, this can be used to interact with the objects in the room to test what they have learnt from the demonstration about collision detection. Firing this projectile at an object will cause it to stick to it once it has detected that it has collided.

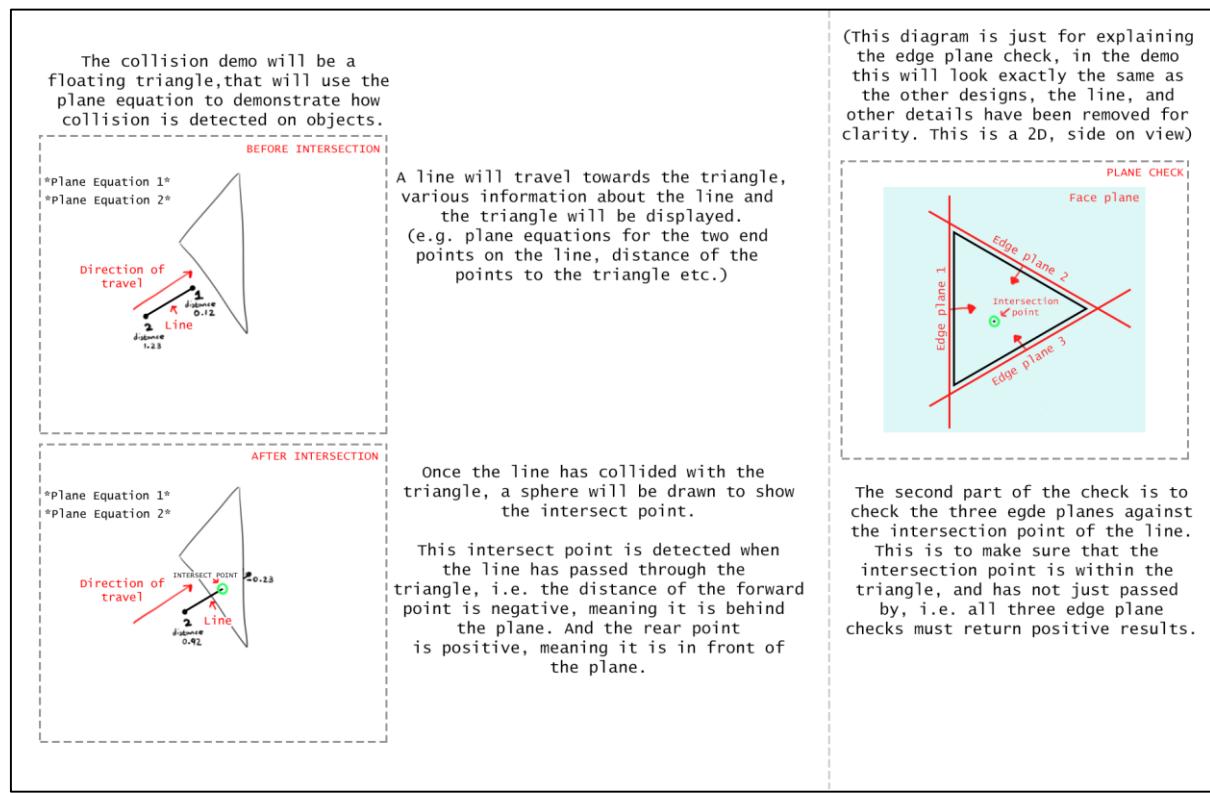


Figure 9 - Collision Demonstration

Figure 9 above shows in detail the demonstration for the collision room. The collision demo will feature a floating triangle and will use the plane equation to demonstrate how collision is detected on objects. This demonstration will use the Unreal level sequencer.

A line will travel towards the triangle, various information about the line and the triangle will be displayed. (E.g. plane equations for the two end points on the line, distance of the points to the triangle etc.)

Once the line has collided with the triangle, a sphere will be drawn to show the intersect point. This intersect point is detected when the line has passed through the triangle, i.e. the distance of the forward point is negative, meaning it is behind the plane. And the rear point is positive, meaning it is in front of the plane.

The second part of the collision check is to check the three edge planes against the intersection point of the line. This is to make sure that the intersection point is within the triangle, and has not just passed by, i.e. all three edge plane checks must return positive results.

This level sequenced demonstration will be accompanied by text, describing in more detail each stage of the process, pausing the demonstration to offer more information as each thing happens.

### 5.5.2.2. Cloth Demonstration

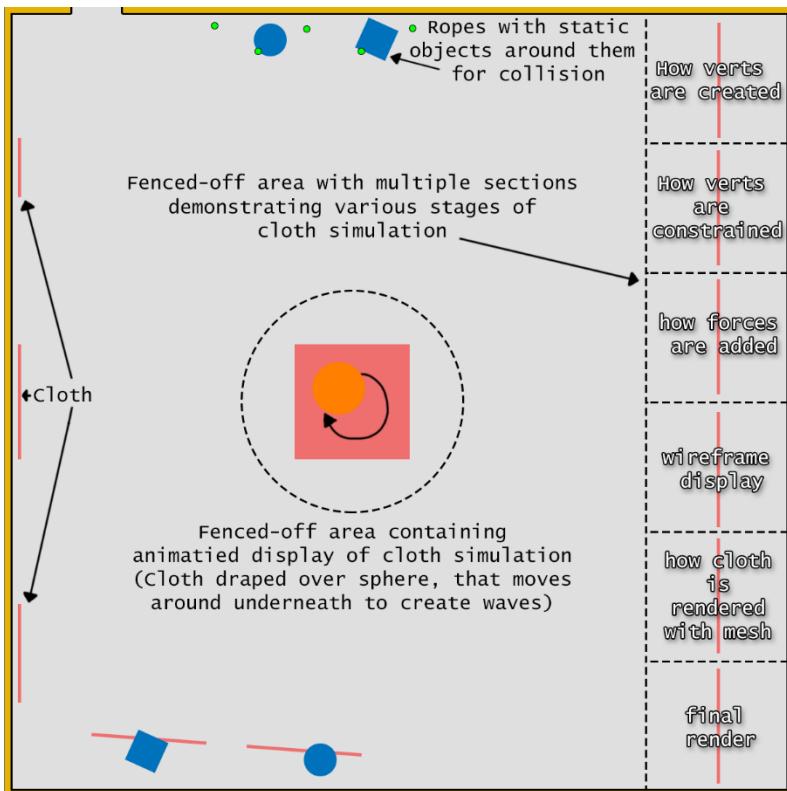


Figure 10 - Cloth Building

Figure 10 above shows the second demonstration in the physics area. This is the cloth simulation demonstration area – the building marked '2' in figure 7.

As with the previous room, the demonstration is located in the middle of the room. This will be a display of the finished cloth simulation. There will be a cloth draped over a sphere, or multiple spheres, and these shapes will move around to demonstrate how realistically the cloth acts. This is explained in further detail in figure 11 below.

To the right side of the room will be a series of small demonstrations lined up next to each other, taking up the whole side of the room. These demonstrations will show the player the different stages of creating a simulated cloth. Firstly, it will explain how the vertices of the cloth are created, next it will explain how these vertices are constrained and held together so they do not fall apart. Thirdly, the application will show how forces such as wind and gravity are applied to the cloth. Then there will be a display of the cloth so far once all of these stages are completed, it will be shown in wireframe like the previous cloths. After this will be an explanation on how a mesh is added to the cloth to give it a realistic appearance, and finally, the finished cloth will be displayed, being blown around by the wind.

There will also be a small display at the top of the room for ropes, these ropes will be created in exactly the same way as the cloths, so a full demonstration is not required again. Instead, just a simple piece of text will be shown describing the process of creating the ropes.

The rest of the room will be filled with various cloths and objects. The cloths will be blown around by the wind, and will collide with the various objects placed near them, further demonstrating the effectiveness of the cloth simulation.

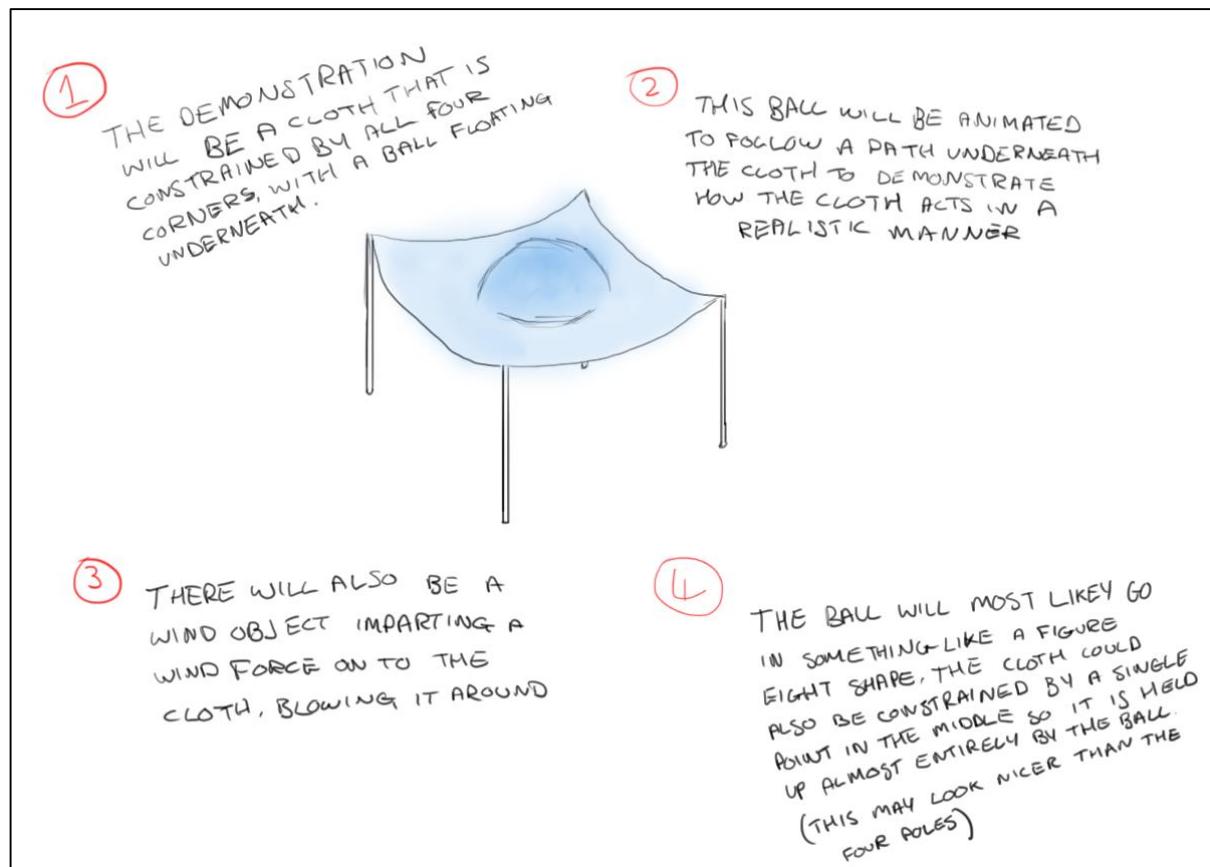


Figure 11 - Cloth Demonstration

The demonstration in the middle of the cloth simulation room will be of a cloth, connected to poles at all four corners, draped over a sphere, or multiple spheres. The spheres will move around underneath the cloth to demonstrate how the cloth moves around when agitated, and how realistic of a cloth simulation can be achieved. The level will also have wind, so this cloth, and all others in the simulation will be blown around by any wind that enters the room from the doors and windows. The spheres will move around on the x, y and z axis, but will always stay underneath the cloth.

### 5.5.2.3. Molecular Simulation Demonstration

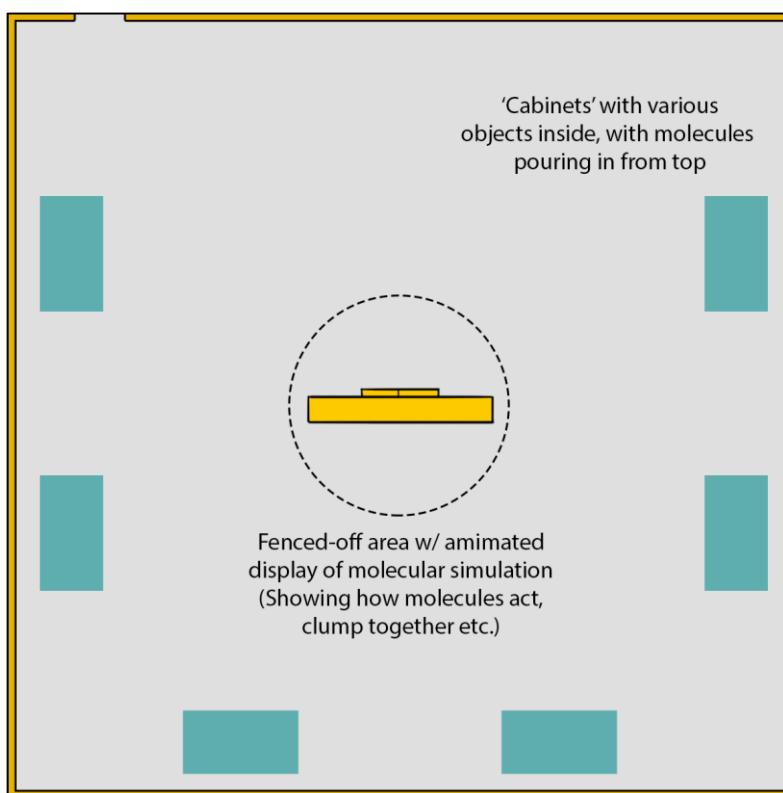
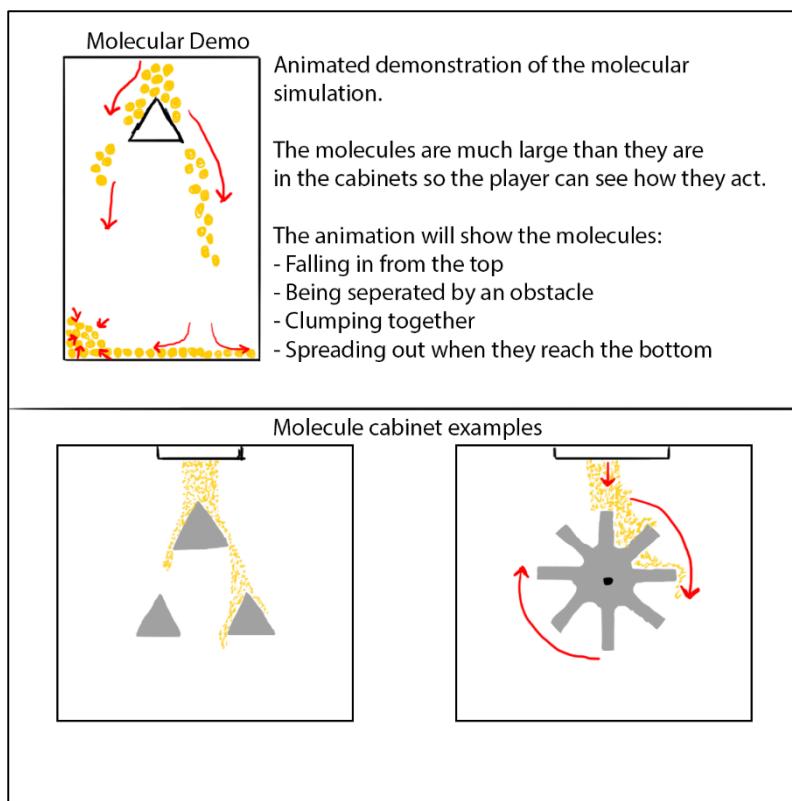


Figure 12 - Molecular Simulation Building

Figure 12 above shows the design for the molecular simulation demonstration area – the building marked ‘3’ in figure 7.

As with previous demonstrations, the main display is located in the middle of the room. This display is similar to the collision room demonstration – using the level sequencer, explaining the various rules that are applied to the molecules when performing the simulation. This demonstration is explained in further detail in figure 13 below.

The walls the demonstration room will be lined with small cabinets, not dissimilar to arcade machines, containing various moving parts and obstacles. These cabinets will fill with the simulated sand or water when the player moves near them; pouring in from the top, the molecules will clump together and move around as if they were a single body of water or sand. Possible examples of these cabinets are shown and explained in further detail in figure 13 below.



*Figure 13 - Molecular Simulation Demonstration*

Figure 13 above shows the molecular demonstration that will be controlled by a level sequence, and two examples of what could be contained within the simulation cabinets.

The animated demonstration of the molecular simulation will not perform any actual simulation, it will simply be an animated display. The molecules in this display will be quite large so they are visible and the effects they have can be shown to the player. The display will show the molecules in a situation, they will fall from the top of the display, fall onto an obstacle and separate, steaming down either side, they will clump together realistically, and spread out and settle at the bottom. This display is essentially a zoomed in version of a cabinet simulation, but with no actual simulation involved.

The cabinet demonstrations will contain possibly thousands of molecules, and will all be in real time. The molecules will pour in from the top of the cabinet, and each cabinet will have a different object or obstacle in it. The molecules will interact with the objects inside the cabinets until all of them have fallen to the bottom and settled. The example on the right on figure 13 shows a gear type wheel. The molecules fall in from the top causing the wheel to rotate, and the molecules to fall to the bottom.

There are some concerns that the system will not be able to handle this, the simulation will have to be done very carefully, maybe even in a separate level, in order to work efficiently. The application will have to perform thousands of collision checks in order to simulate this, but there are ways this could be optimised, for example splitting molecules up into groups, and only performing collision checks on molecules in the surrounding groups. This would significantly reduce the number of checks required. Only time will tell however, once the simulation is implemented, a better understanding of its feasibility will be gained.

#### 5.5.2.4. Flocking Demonstration

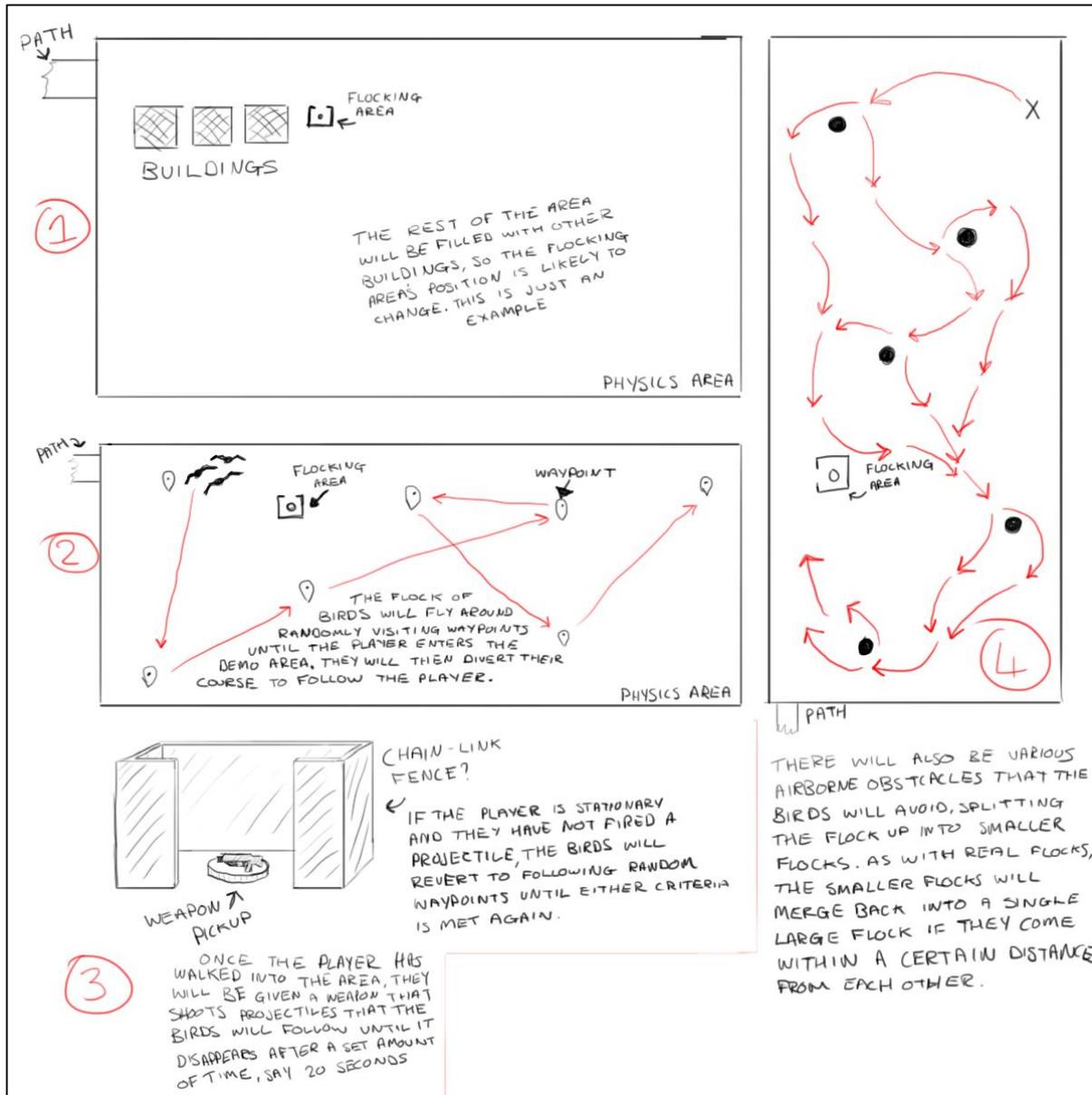


Figure 14 - Flocking Area

Figure 14 above shows the design for the flocking demonstration area of the level. The starting point for the flocking area is the area labelled '4' in figure 7.

This demonstration is located outdoors because of its size. The demonstration will begin when the player moves into the weapon pickup area marked with a red 3 in figure 14 above. Once the player has walked into the area, a number of birds will spawn and start flying around randomly, flocking together like real birds. The flocks of birds will fly around randomly visiting waypoints. When two or more birds get close to each other, they will group together forming flocks. The flocks will eventually group up into large groups of birds. There will be various airborne obstacles in the way of the birds. When a flock encounters an obstacle, the birds will try and avoid it, this will split the flock up into multiple flocks, when the birds get close together again, they will re-join the flock.

The player will also be equipped with a weapon that first two kinds of projectile; one the birds avoid like an obstacle, and one they follow as if it were another bird. These projectiles will have a set lifetime, once the projectile disappears, the birds will continue to fly around randomly.

This demonstration will be accompanied by a textual description of the flocking mechanics, and what rules are needed to implement it effectively.

### 5.5.3. Lighting and Effects Area

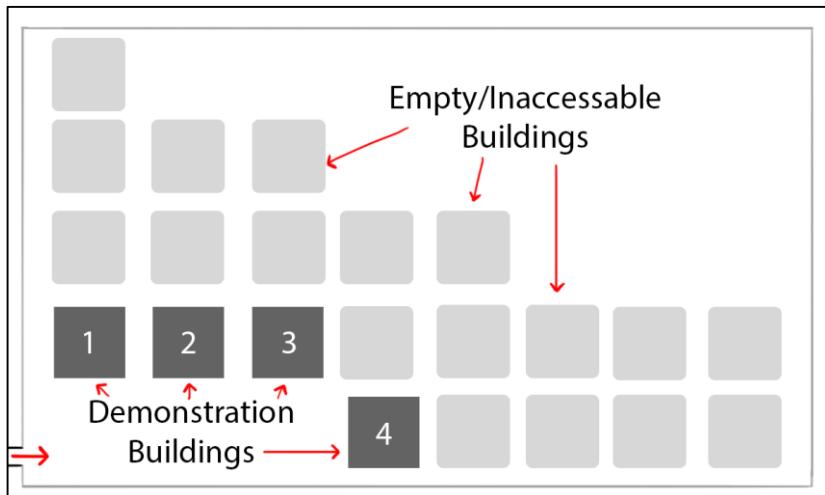


Figure 15 - Lighting/Effects Area Overview

Figure 15 above shows the layout for the lighting and effects area - labelled in figure 4 as 'Lighting'. The lighting and effects area will have four demonstration areas, these will be located in four buildings. There will also be a variety of inaccessible buildings acting as scenery. The entrance to the area is marked on the design as a red arrow in the bottom left. All demonstrations in the lighting and effects area will be accompanied by textual displays, explaining everything that is happening in the demonstration.

#### 5.5.3.1. Types of Light, Shadow, and Lighting Effects Demonstration

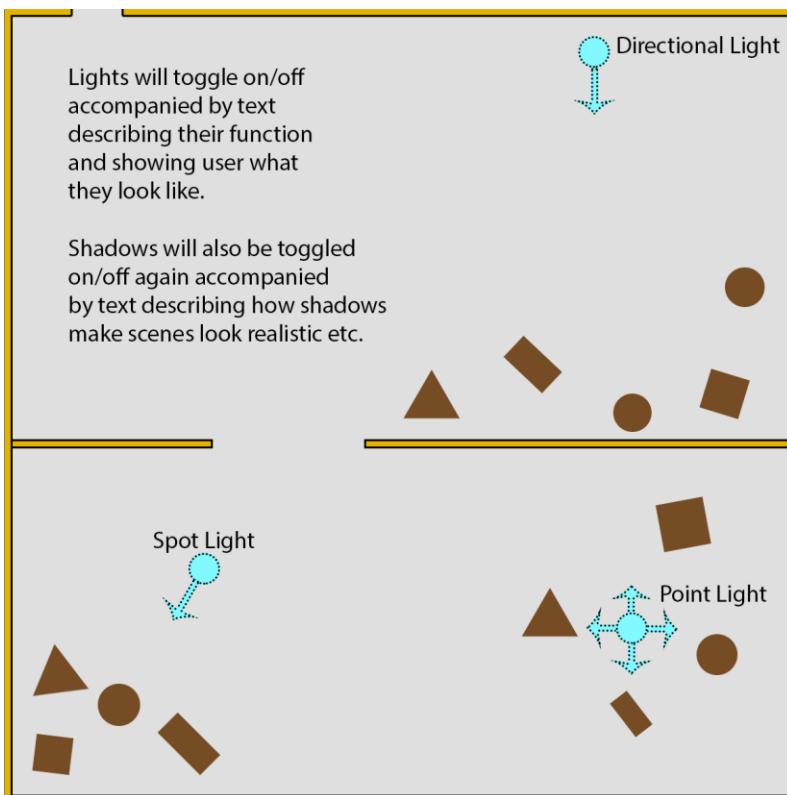


Figure 16 - Types of Light Building

Figure 16 above show the first of the demonstrations in the lighting and effects area. This is the types of lights, shadows and lighting effects demonstration area – the building marked ‘1’ in figure 15.

The windows of this demonstration room will be covered so as little outside light as possible enters the room. This is to ensure that it does not interfere with the demonstration of the lights. The room will be filled with various objects, and three of the four lights available in Unreal will be placed at different locations in the room pointing at a set of objects. The fourth light that is not included is the sky light, which cannot be effectively demonstrated outside of simply looking outside the building and seeing that it is light.

Each of the lights will be toggled off and on, cycled through etc. in order to show what each light does, and what effect it can achieve. The shadows will also toggled off and on to show what effect shadows have on a scene. As with other demonstrations, this will be accompanied by a textual description.

#### *5.5.3.2. Post Processing Effects Demonstration*

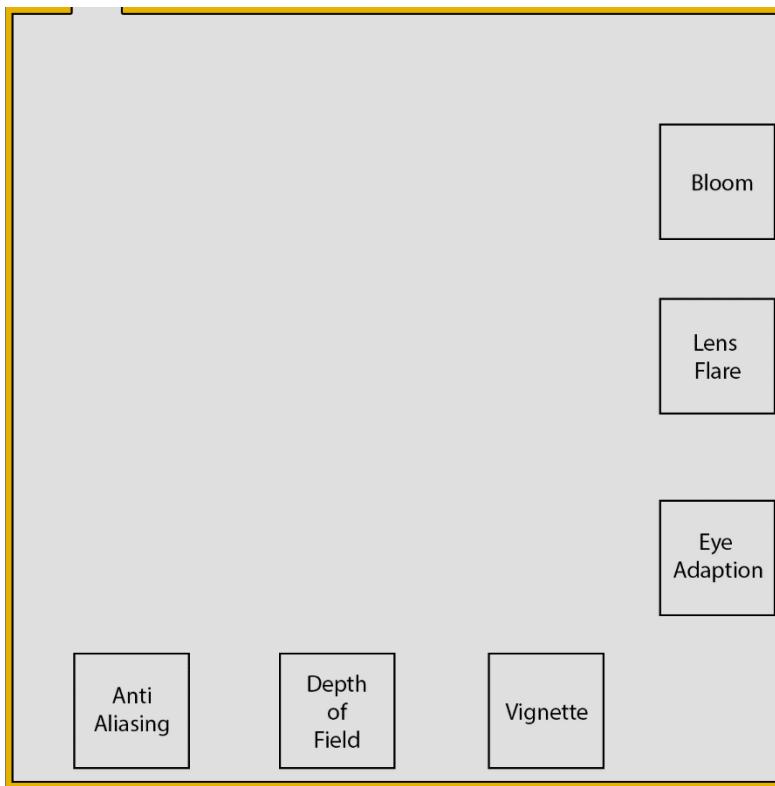


Figure 17 - Post Processing Building

Figure 17 above show the second demonstration in the lighting and effects area. This is the post processing effects demonstration area – the building marked ‘2’ in figure 15.

The post processing effects demo will be fairly simple. The room will be arranged with six sub-areas, each containing the demonstration for an individual post processing effect. Each effect will be turned on, and text will appear describing it, the effect will then turn off and on to show a comparison between no effects vs. effects. Figure 18 below goes into more detail for each demonstration.

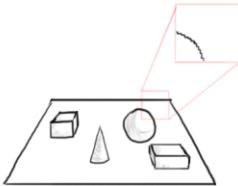
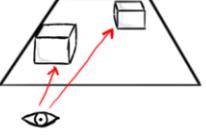
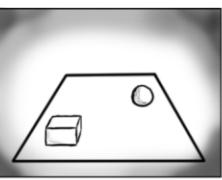
Anti-Aliasing	Depth of Field	Vignette
		
Variety of objects. anti-aliasing toggled off/on to show difference between effect and no effect	Two objects: one near, one far. DoF will toggle off/on to show difference between effect and no effect	Vignette will toggle off/on to show difference between effect and no effect
Eye Adaption	Lens Flare	Bloom
Light will be toggled off/on to show what eye adaption looks like  	Light will be toggled off/on to show what lens flare looks like  	Light will be toggled off/on to show what bloom looks like  

Figure 18 - Post Processing Demonstrations

Figure 18 above shows each of the six demonstrations for the six post processing effects. The anti-aliasing demonstration will contain a variety of objects, anti-aliasing will be toggled off and on to show the difference between anti-aliasing and no anti-aliasing. The player will be able to notice jagged edges on the objects in this demo.

The depth of field demonstration will contain two objects, one close to the player (foreground), and one far away from the player (background). The depth of field effect will then be toggled off and on to show the player the difference between DoF and no DoF. When DoF is on the player should be able to notice that the object in the foreground that they are looking at is sharp and in focus, and the object in the background that is blurry.

The vignette demonstration will contain a couple of objects, they will however, not be very important to the demonstration. The vignette effect will toggle off and on to show the player the difference between vignette and no vignette. The player should notice dark edges around the screen when vignette is enabled.

The eye adaption demonstration will only contain a single light source. This light will be toggled off and on to show the eye adaption effect in play. When enabled, the player should be able to notice that the brightness of the level seems to change automatically based on how bright the thing they are looking at is. This is to simulate the human eye adapting to differing brightness levels.

The lens flare demonstration will only contain a single light source. This light will be toggled off and on to show the lens flare effect in play. When enabled, the player should be able to notice shapes on

the screen revolving around the light source when the camera moves. This is to simulate the lens flare phenomenon that occurs when using cameras.

The bloom demonstration will only contain a single light source. This light will be toggled off and on to show the bloom effect in play. When enabled, the player should be able to notice that lights seem brighter, this is to simulate the haze that is felt when looking at really bright objects.

#### 5.5.3.3. Materials and Material Properties Demonstration

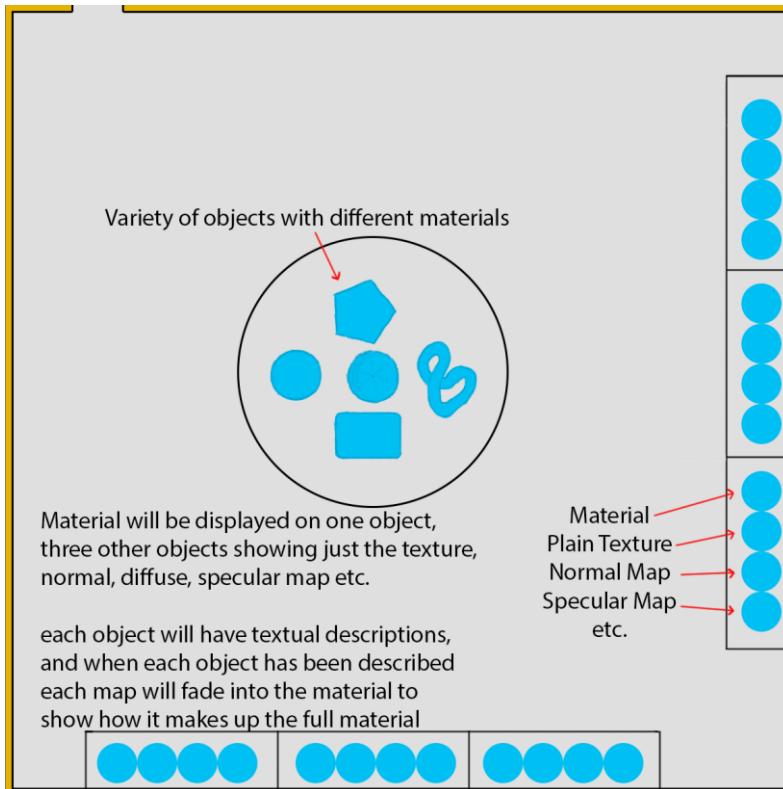


Figure 19 - Materials Building

Figure 19 above show the third demonstration in the lighting and effects area. This is the materials and material properties demonstration area – the building marked '3' in figure 15.

This demonstration will have a variety of objects, textured with different materials all laid out across two of the walls of the demonstration room. These materials will have different properties, such as roughness and metallicness and each material will also have various texture maps, such as normal maps.

Each material will be displayed on an object to the player, each will also be accompanied by a display of each texture map the object has. So if a material has a texture, normal map and diffuse map. The material will be displayed on one object, which will be accompanied by three other objects showing just the texture, diffuse, and normal maps that make up the material.

These materials will have textual displays describing what each map does, and how it is used to allude to more detailed materials. The material properties will also be described here, each material changing their properties to show how they affect the final outcome. When each object has been described each map will fade into the material to show how it makes up the full material.

#### *5.5.3.4. Reflections Demonstration*

The final demonstration in the lighting and effects area is the reflections demonstration area – the building marked ‘4’ in figure 15.

There is no set design to this area, it will simply be filled with whatever assets are available at the time of development. Within the demonstration building will be a small scene containing lights, mirrors, reflective surfaces etc. This demonstration will show how reflections are important to a scene, and demonstrate things like screen space reflections. To demonstrate this, reflections will be toggled off and on to show the difference in quality of the scene. The demonstration will also be accompanied by a textual description of reflections and why they are important while the player is walking through the scene.

#### *5.5.4. Sandbox Area*

The sandbox area will be a mixture of both the physics and lighting/effects area. It will be an informal blend of elements from the two areas, there will be little to no explanations or descriptions about the demonstrations.

This area will be more of a playground for players to have fun with the different elements of the application, such as the different weapons, the cloth, or the sand. To ensure the informal style is kept, this area will not have a set design, and will evolve naturally depending on what is added to the application and what assets are available. This area will be made at the end of development, and is not an important aspect; therefore, it can be cut from the application without affecting the core objectives of the software. This is simply an optional area of the application that can be added to improve the entertainment aspect of the application.

## 6. Implementation

### 6.1. Project Creation, Landscape & Whiteboxing

To create the project, the ‘New Project’ dialogue was simply pressed. In the figure below you can see the new project dialogue, this is where every Unreal project starts. There are two tabs: ‘Blueprint’ and ‘C++’. This project will not be using blueprints, everything will be coded in the C++ language, so this is the option that was selected.

As stated in the design documentation, the project will have a first-person view, so the ‘First Person’ template project was selected to allow there to be a foundation to build upon, rather than starting from scratch. The project directory was selected, the project named, and the create button clicked.

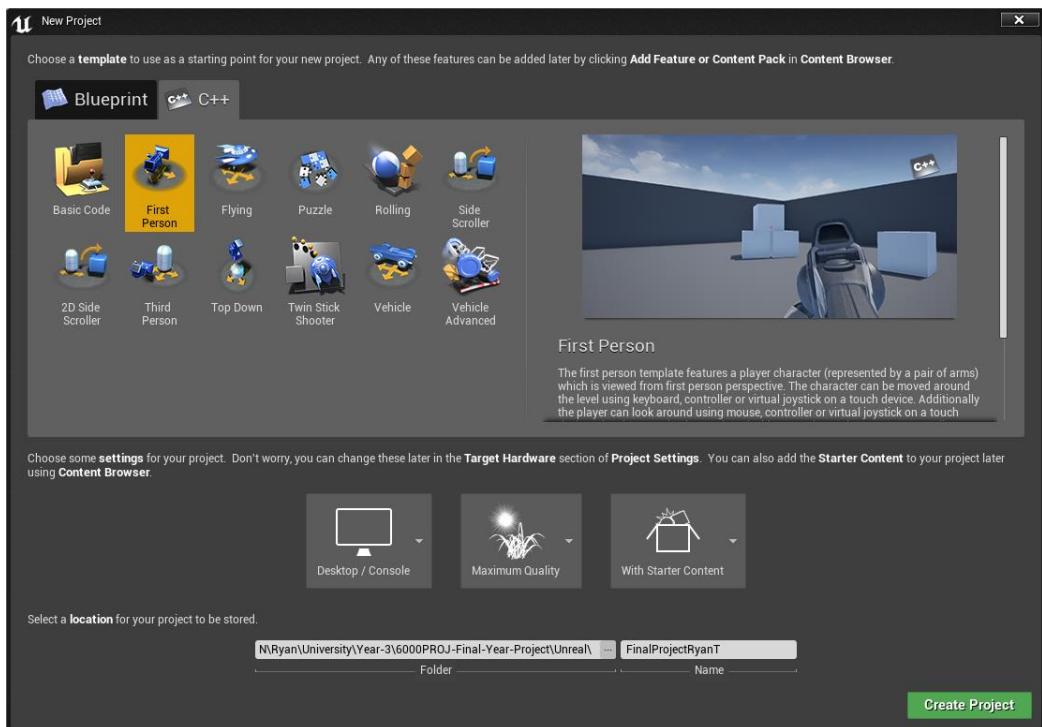


Figure 20 - Project Creation

In the figure below, you can see the created project. A small level has already been created with various boxes and walls, but this is not needed, so a new level will be created.



Figure 21 - Default Level

Below, you can see the new level dialogue. Here a blank level will be created without all of the example content.

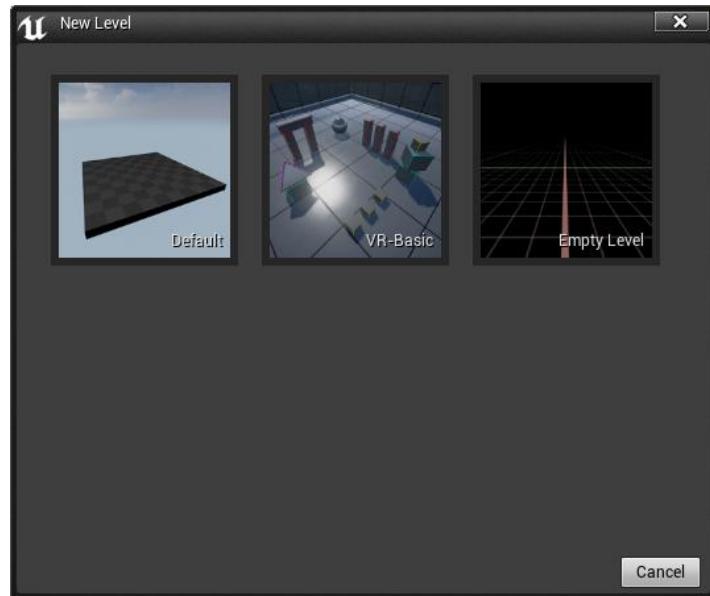


Figure 22 - New Level Dialogue

This is the blank level, it does however have a small floor. This will not be needed because the landscape of the level will be made with the landscape tool, not out of meshes.

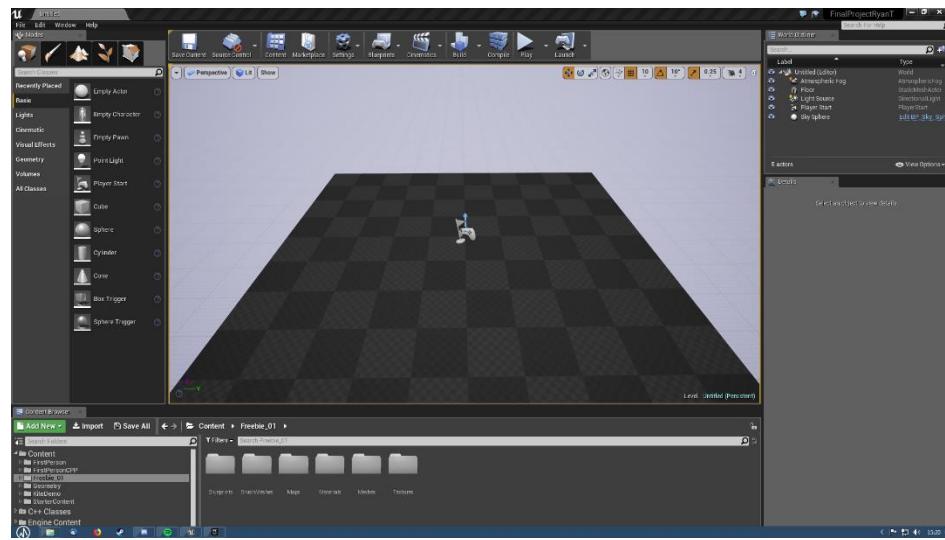


Figure 23 - Blank Level

In the figure below, the landscape editor has been selected, this can be seen in the ‘Modes’ panel on the left side of the screenshot, highlighted in red. The small floor object as seen in the previous screenshot was deleted, and the landscape editor opened.

In the options panel on the left side of the window, ‘Fill World’ was clicked, this increases the landscape object to fill the size of the level, a material was then chosen for the landscape, in this case it was ‘M\_Ground\_Gravel’, but this will act as a placeholder and will be soon changed. After this and other options were tweaked, the landscape was created using the ‘Create’ button.

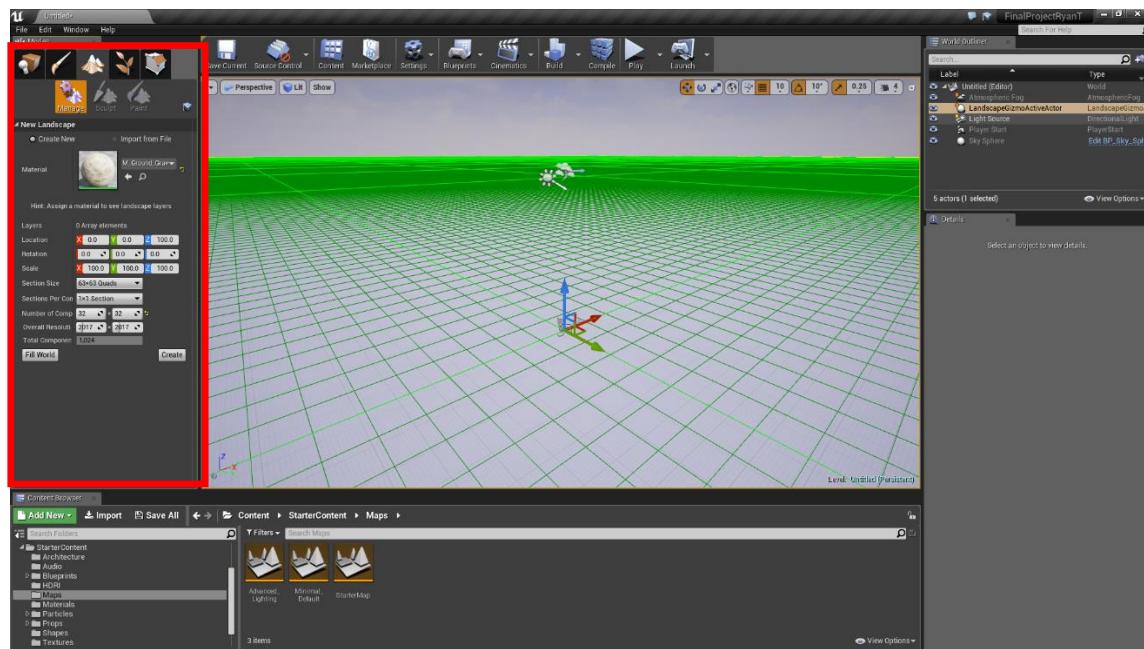


Figure 24 - Landscape Editor

Below you can see the created landscape. You can also see three widgets in the viewport. The selected one in the middle of the viewport is the player controller, this is where the player will spawn when the game is started. On the left/middle, with an icon depicting a sun, is the ‘Light Source’ for the level; this is basically the sun. Finally the one on the right that looks like clouds, is the ‘Atmospheric Fog’ object; this controls various parameters about the atmosphere such as fog. It will also place a sun object in the sky, opposite the main light source, this gives the effect that the light is coming from the sun instead of out of an invisible object. This ‘sun’ will move depending on the position on the light source, and the sky colour will also change based on the height of the ‘sun’.

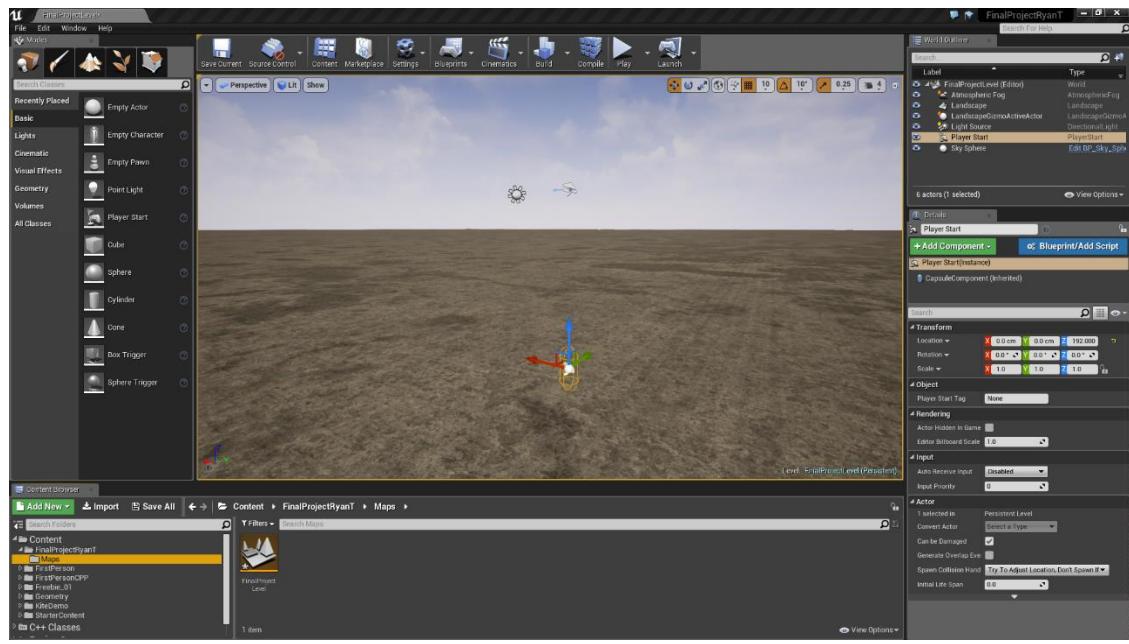


Figure 25 - Created Landscape

Below you can see a ‘Lightmass Importance Volume’ has been added around the landscape. This only renders high quality lighting within the volume. This volume should cover the playable area, in this case, the whole map.

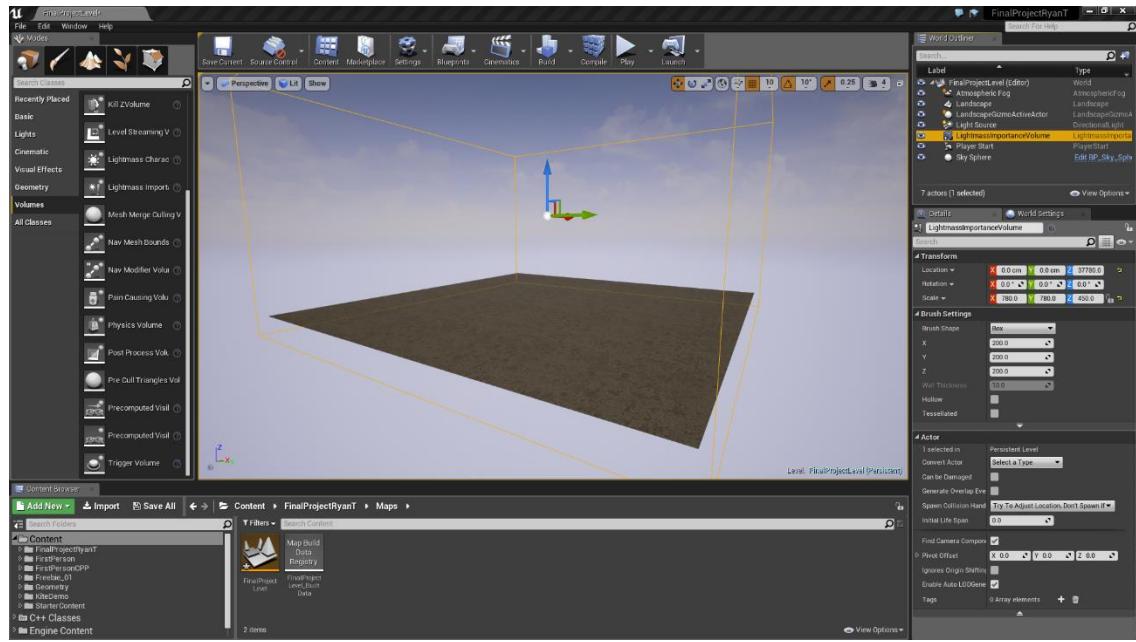


Figure 26 - Lightmass Importance Volume

In the figure below, we can see the start of the ‘whiteboxing’ process, this is the stage of development where the level is boxed out; placeholder objects added to represent the position of the final assets. Following is the process of whiteboxing a fountain for the starting zone.

In the below screenshot a cylinder from the ‘Geometry’ tab of the modes panel has been dragged into the scene and sized appropriately.

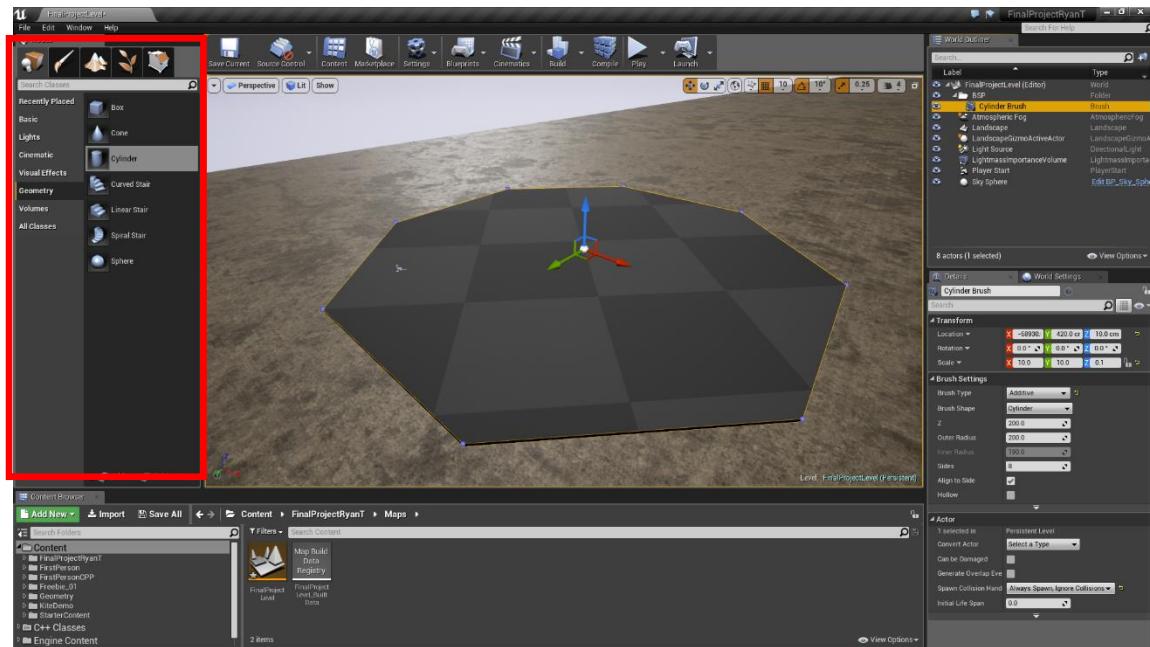


Figure 27 - Start of Whiteboxing

The shape was then made hollow, and the number of sides changed to 20. The shape was made hollow so it acts as a barrier wall rather than a floor.

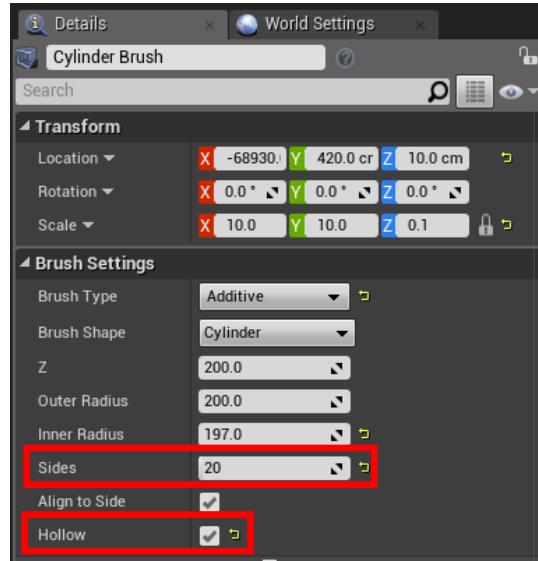


Figure 28 – Brush Settings

Below you can see the completed, whiteboxed starting zone. The same tools were used to make the rest of the fountain.

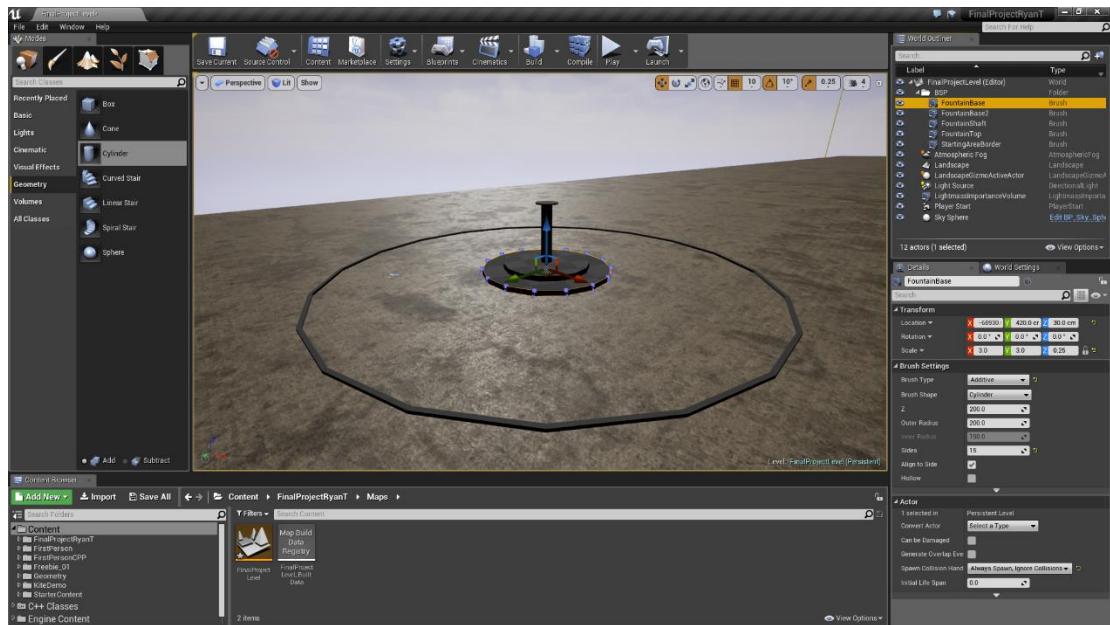


Figure 29 - Whiteboxing Starting Zone Complete

Using the same tools, a box was made and placed over a portion of the border.

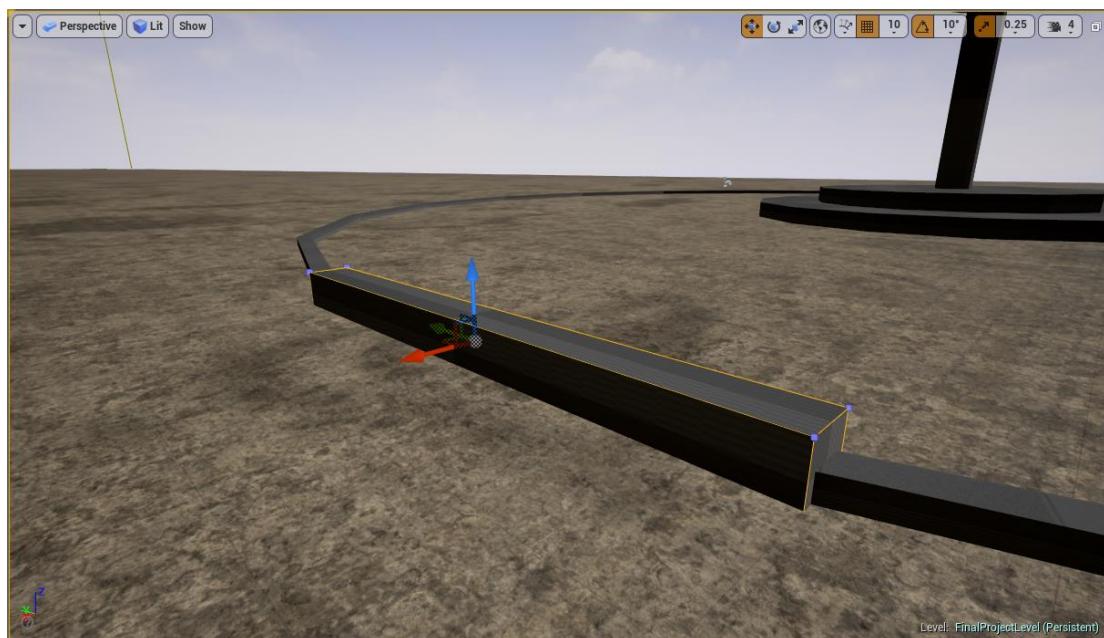


Figure 30 – Placing Geometry

This box was then set to ‘Subtractive’, this changes it from a piece of geometry, to something that removes geometry.

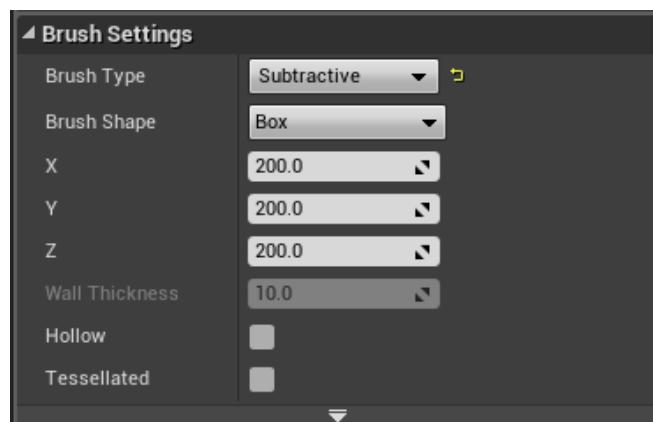


Figure 31 - Brush Settings

Here you can see the subtractive geometry in action. This cut-out will be used to hold a path to one of the areas.

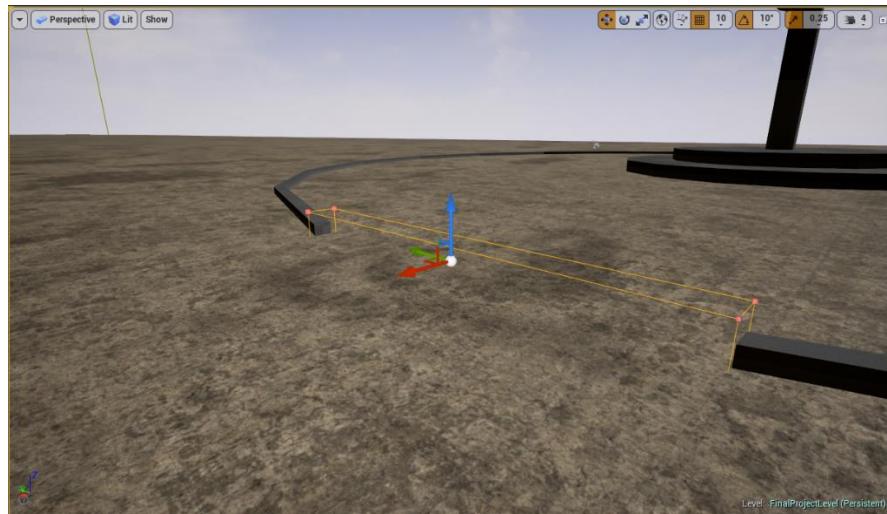


Figure 32 - Subtracted Geometry

Using sculpt tool to form landscape.

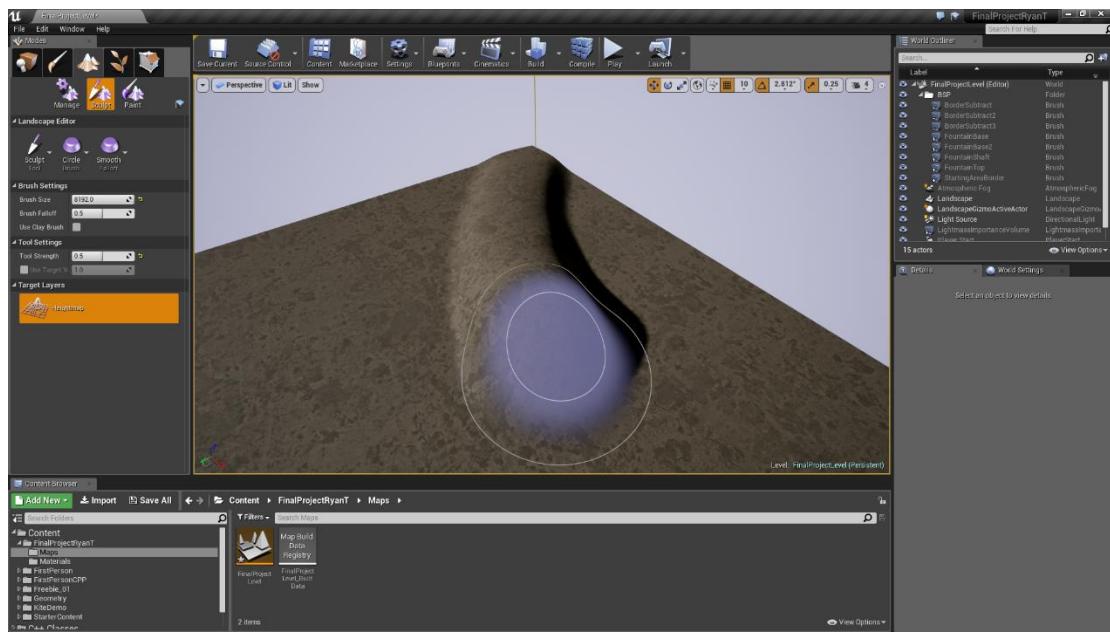


Figure 33 - Start of Landscape Sculpting

Continuing to use the sculpt tool, mountains are being made

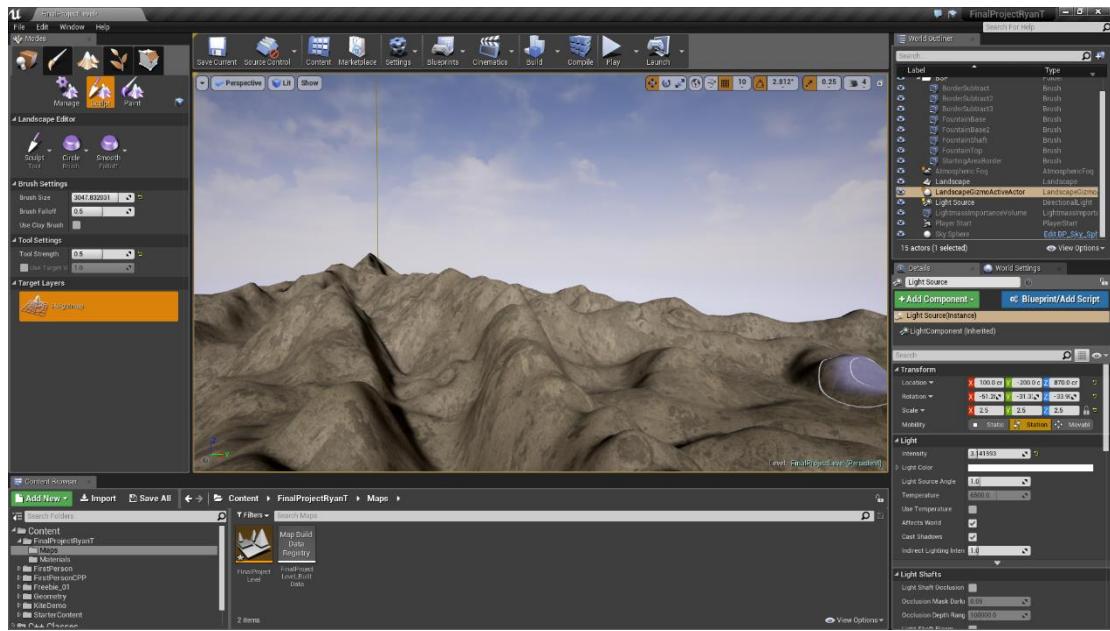


Figure 34 - Starting to Make Mountains

Rough sculpting progress

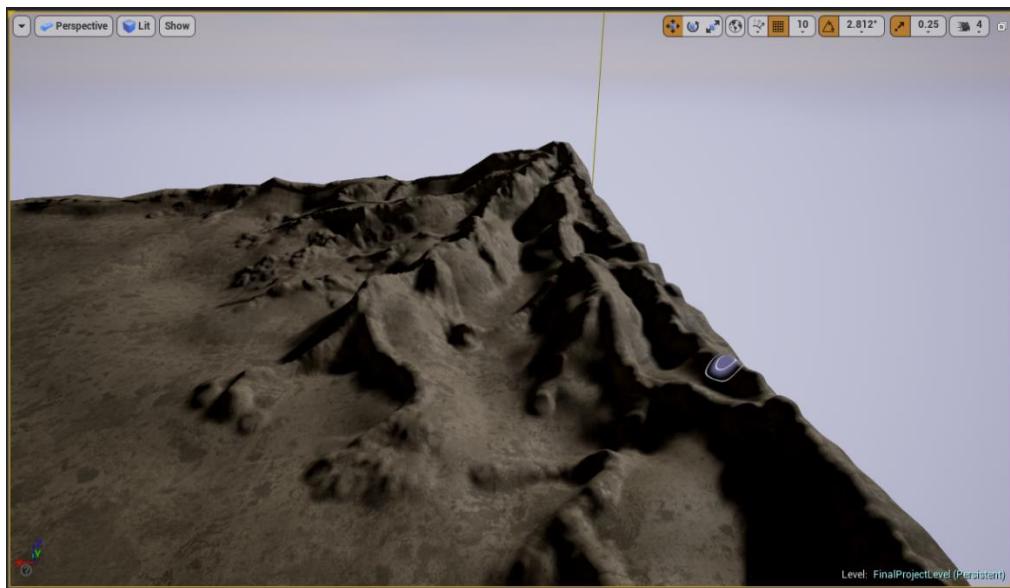


Figure 35 - Rough Mountains Progress

Rough sculpting progress (Top down)



Figure 36 - Sculpting Progress Overview

Smooth tool before

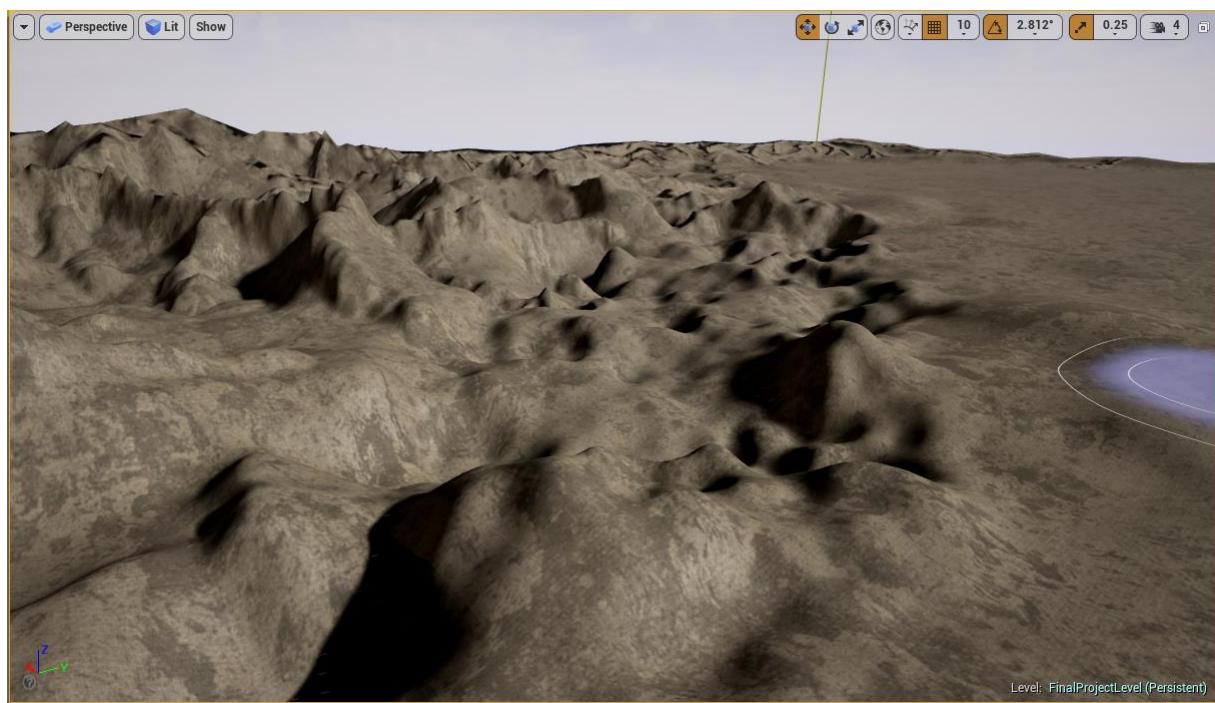


Figure 37 - Before Smooth Tool

## Smooth tool after

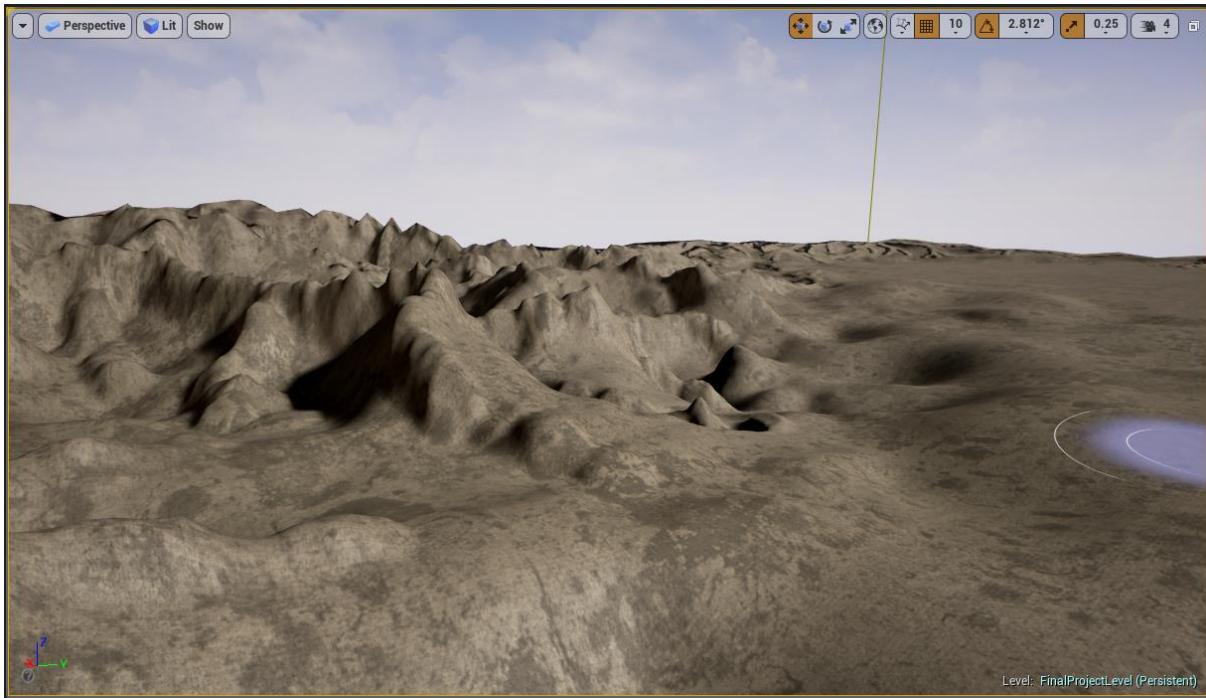


Figure 38 - Smooth Tool After

## Erode tool before

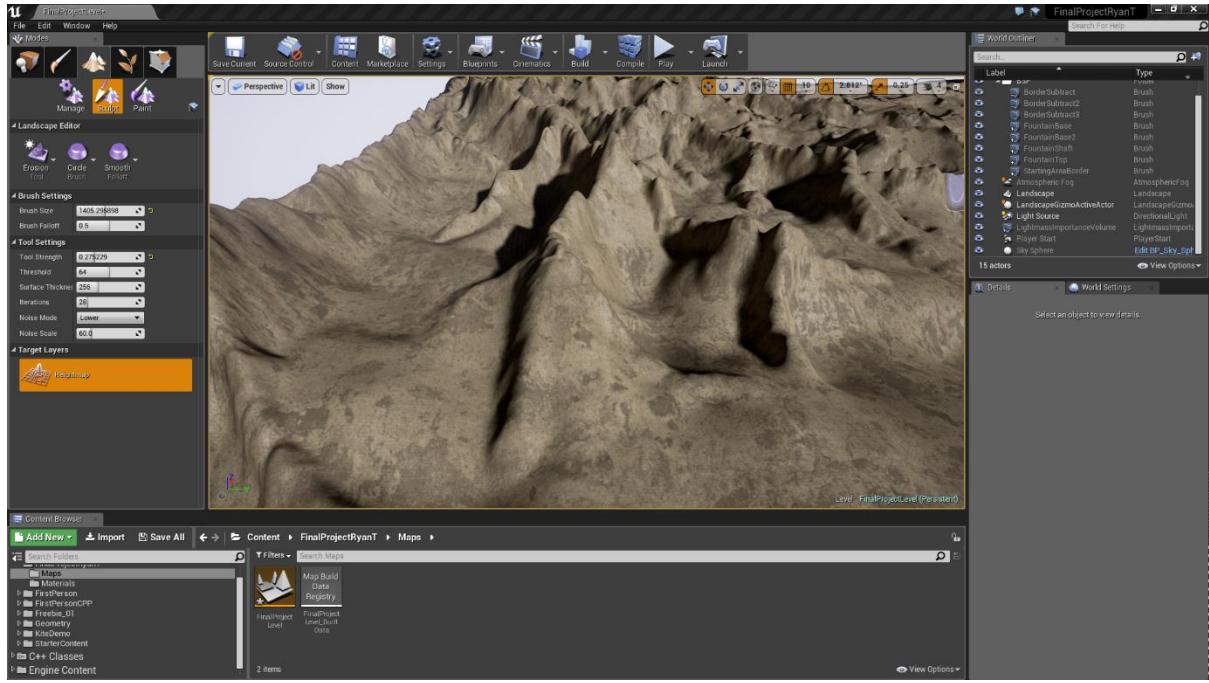


Figure 39 - Erode Tool Before

## Erode tool after

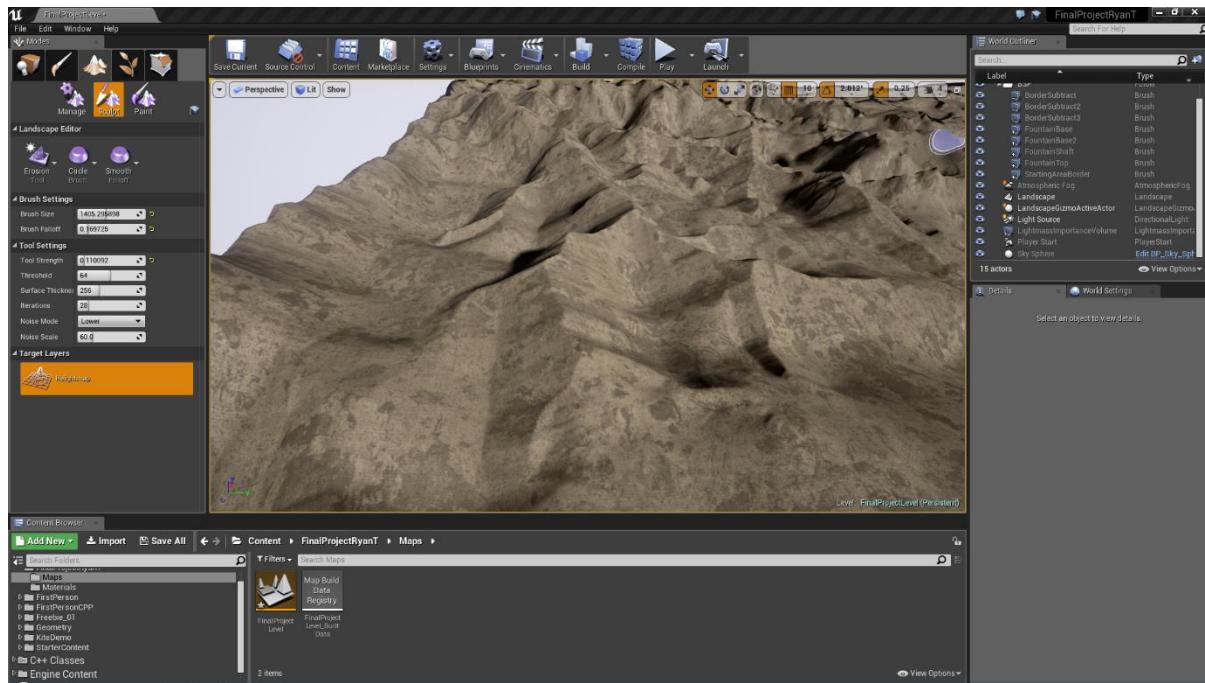


Figure 40 - Erode Tool After

## Erode tool before 2

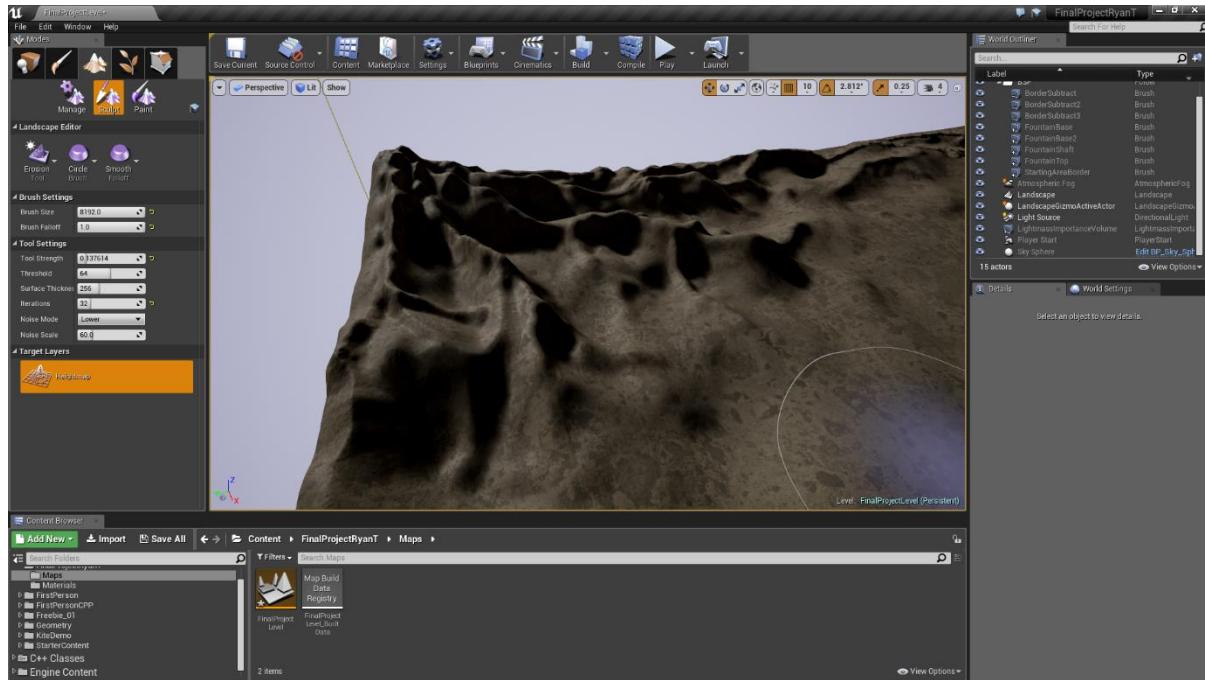


Figure 41 - Erode Tool Before

## Erode tool after 2

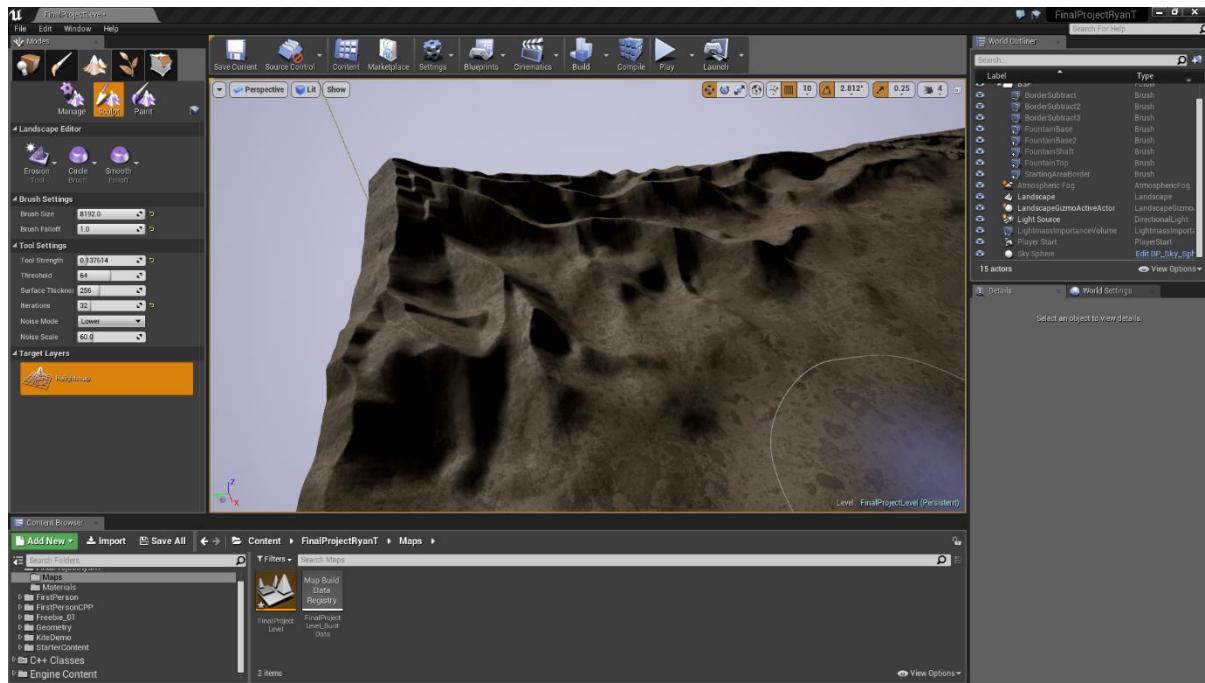


Figure 42 - Erode Tool After

With no texture, squares are stretched, meaning texture will be too.

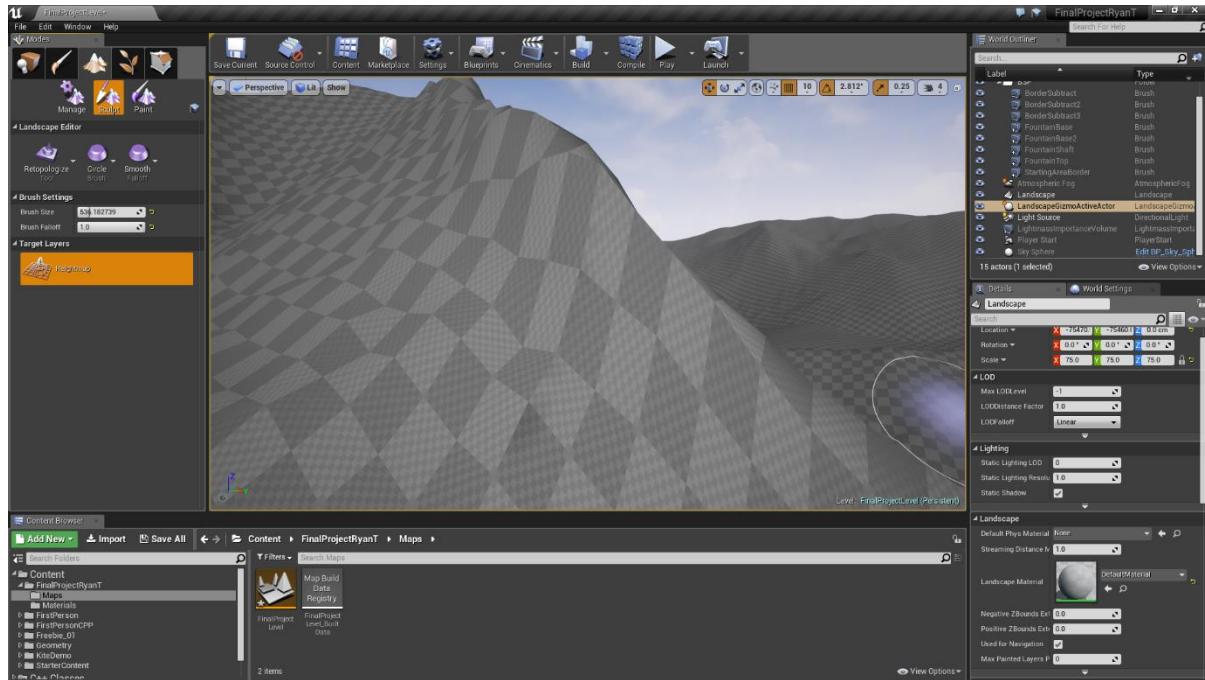


Figure 43 - Stretched Textures

Using retopologise tool, squares are smoothed out and no longer stretched

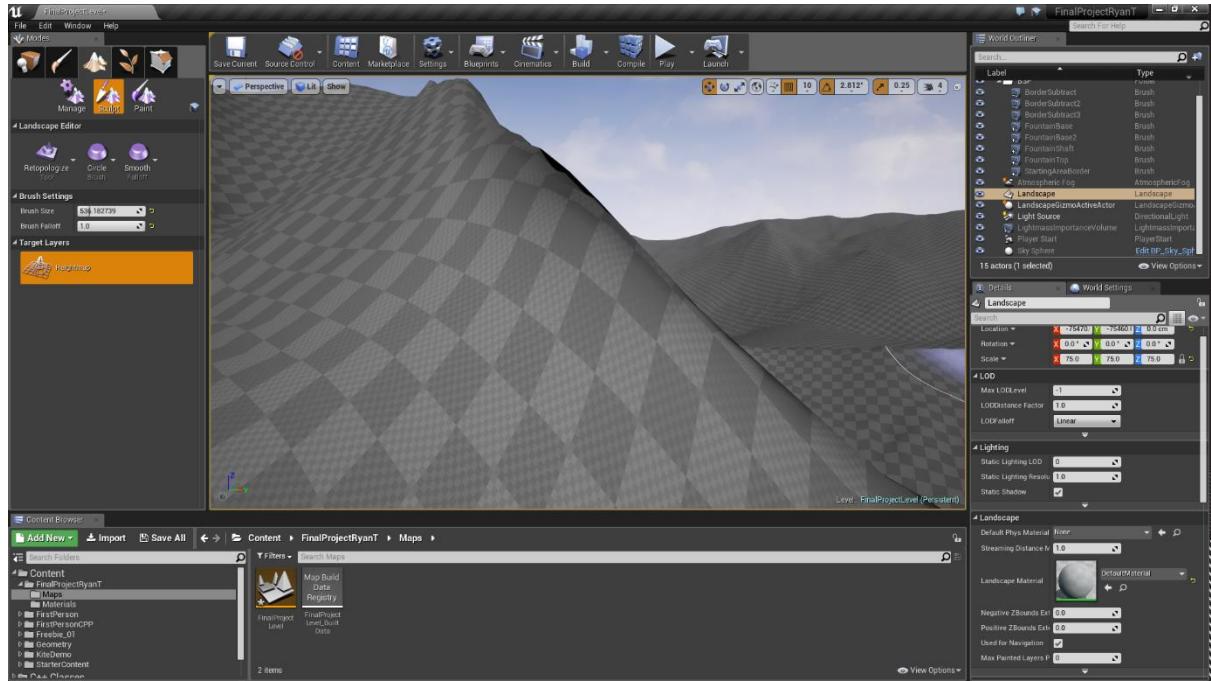


Figure 44 - Retopologise Tool

Using same tools as with starting zone, the three areas have been whiteboxed.

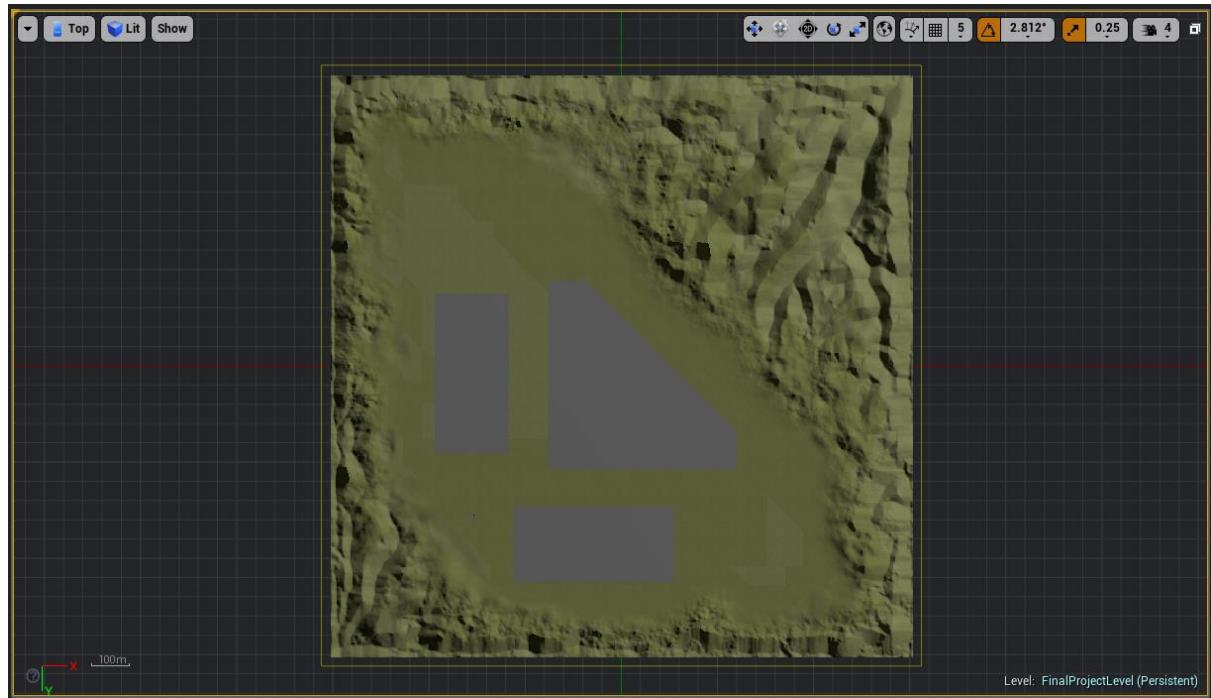


Figure 45 - Whiteboxed Level Areas

New material is needed for landscape, a new directory called ‘Materials’ was created, and by right clicking in the content browser and selecting new material a new material can be created.

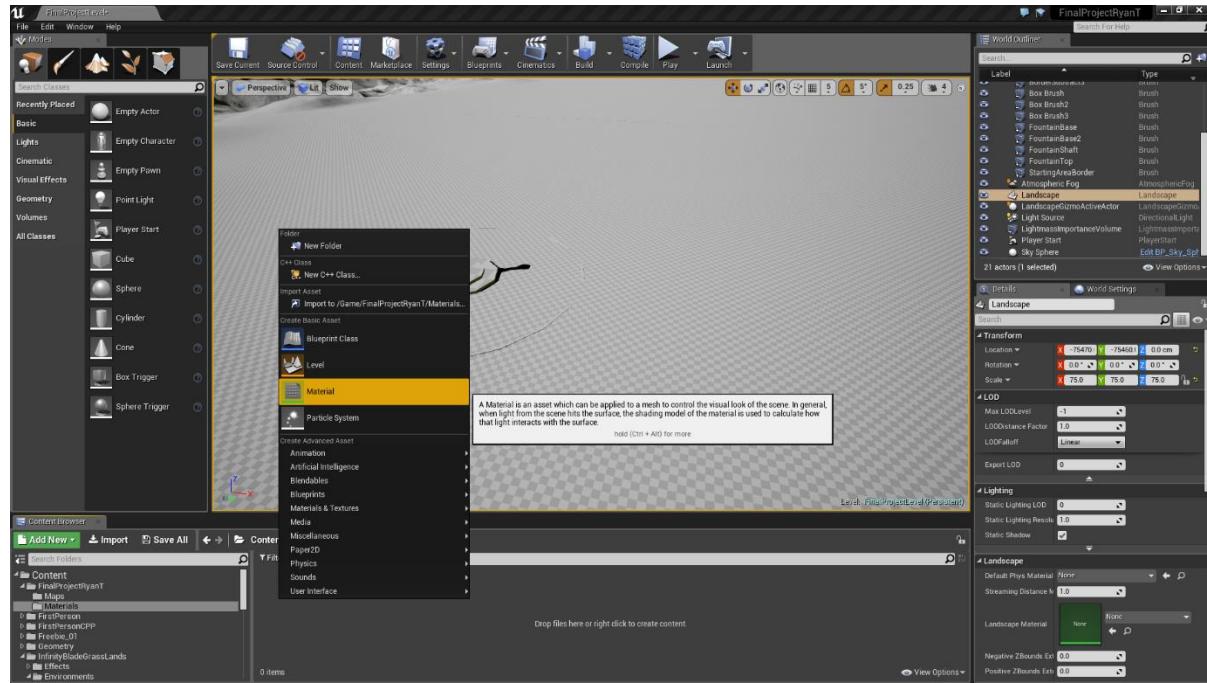


Figure 46 - New Material

Below you can see this new material has been created

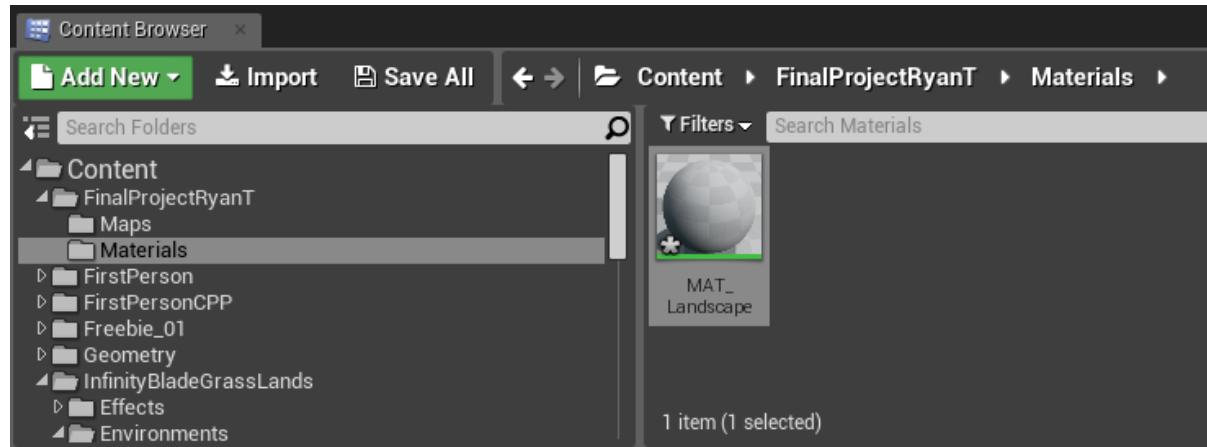


Figure 47 - New Material in Content Browser

## This is material editor

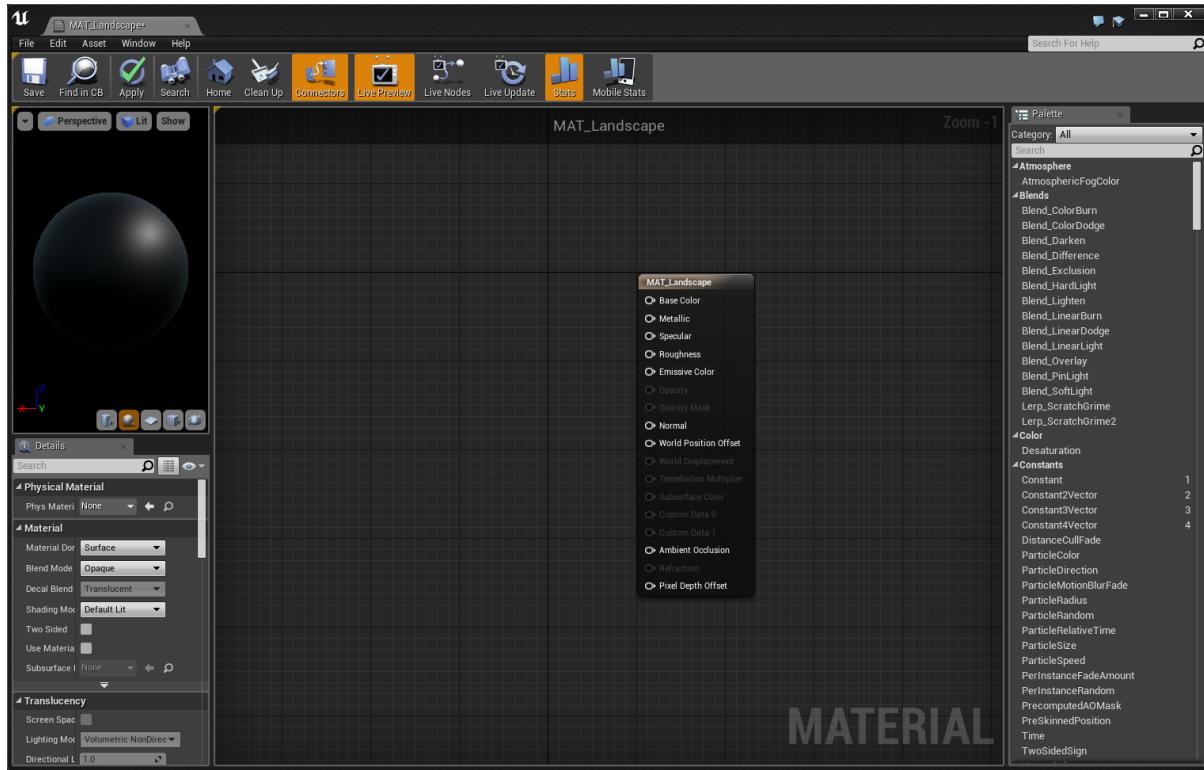


Figure 48 - The Material Editor

Added first texture

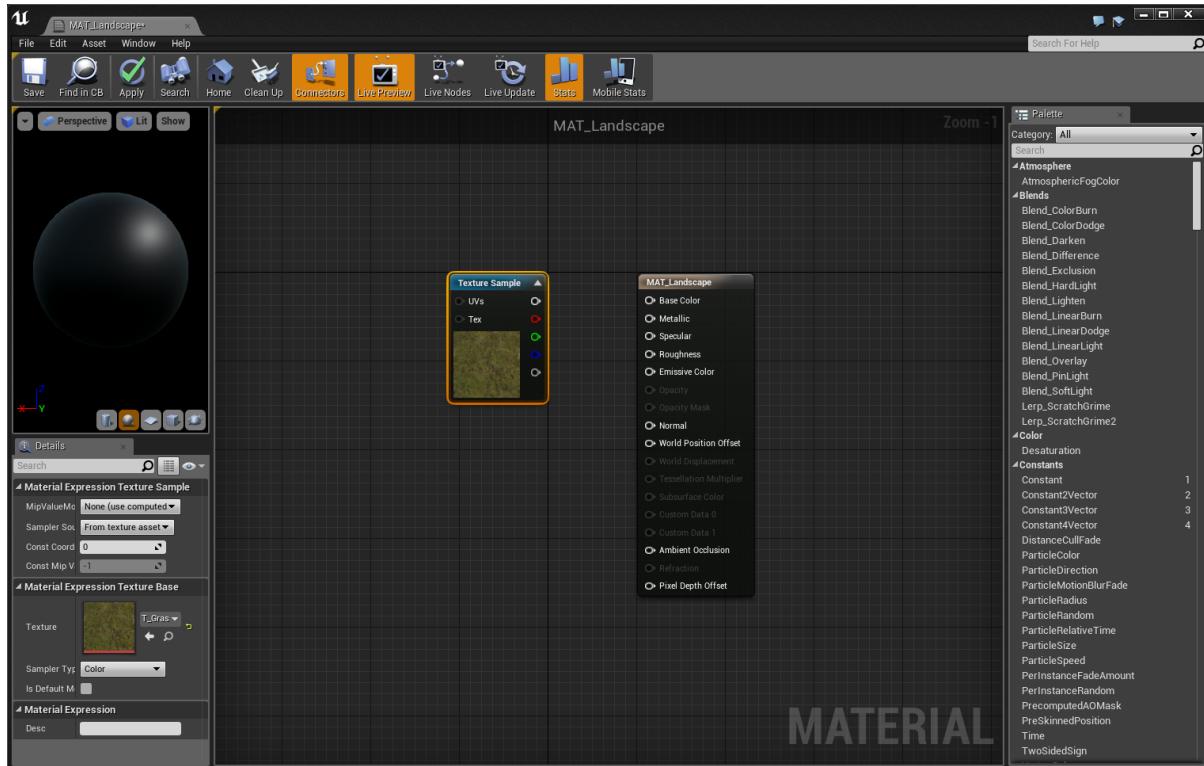


Figure 49 - First Texture Added

Connected the white node from texture to ‘Base Color’, this is the channel for all three colour channels (Red, Green & Blue).

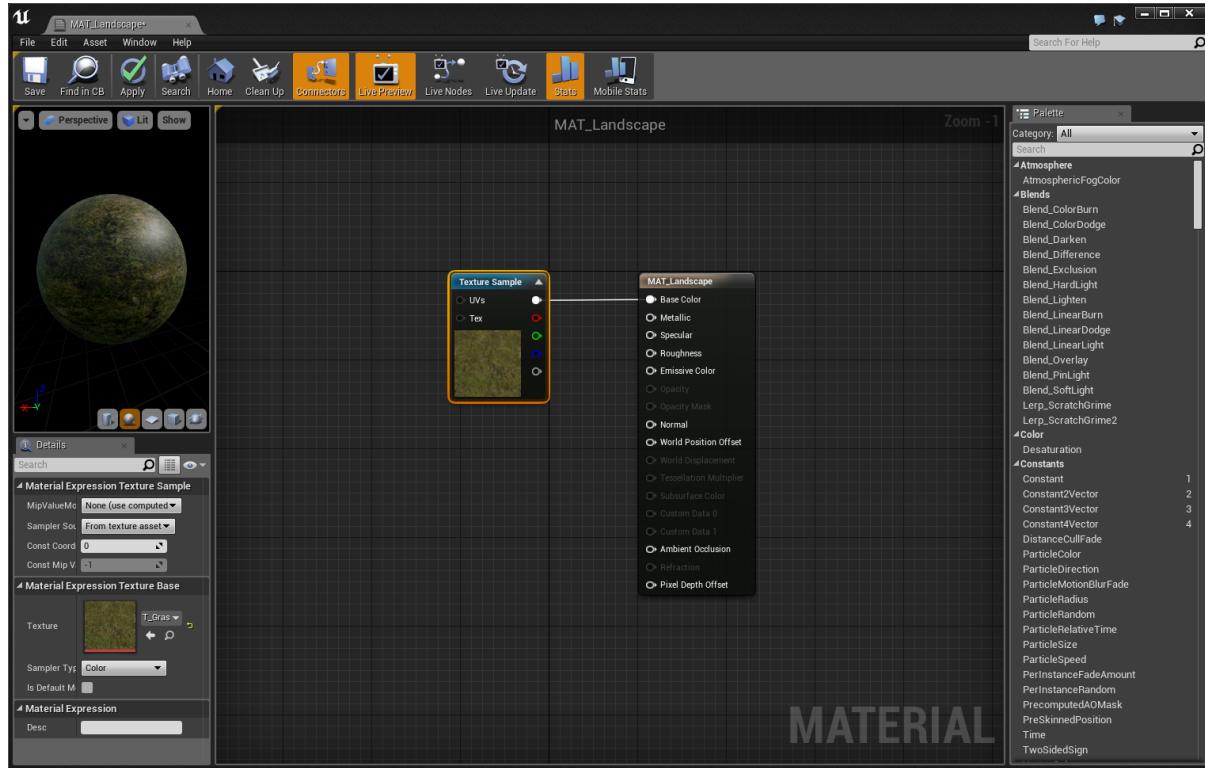


Figure 50

Added the normal map for the texture, and connected the white node to ‘Normal’. Normal maps are used to give depth to a texture and affect how light bounces off it, making it look more tessellated instead of just a 2D image.

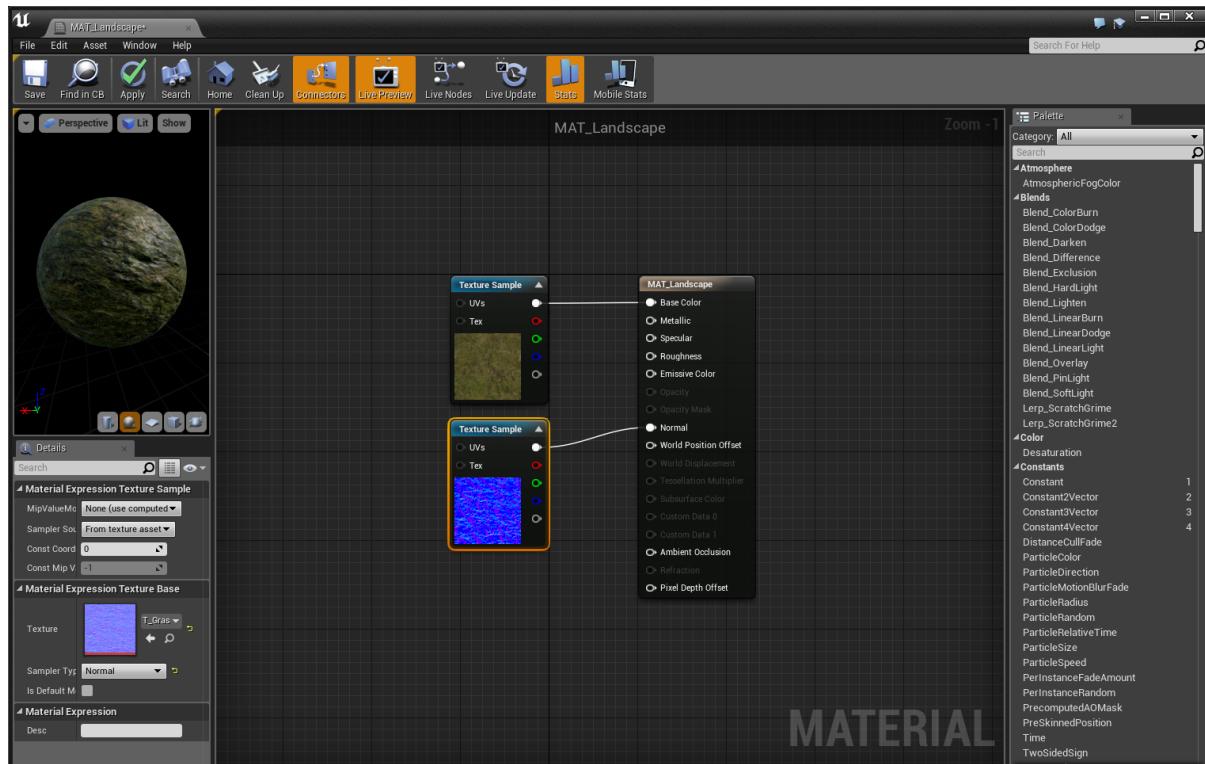


Figure 51

Texture looked too shiny, so a constant value of 0 has been added to the 'Metallic' node of the material. Taking away all metallicness for the texture.

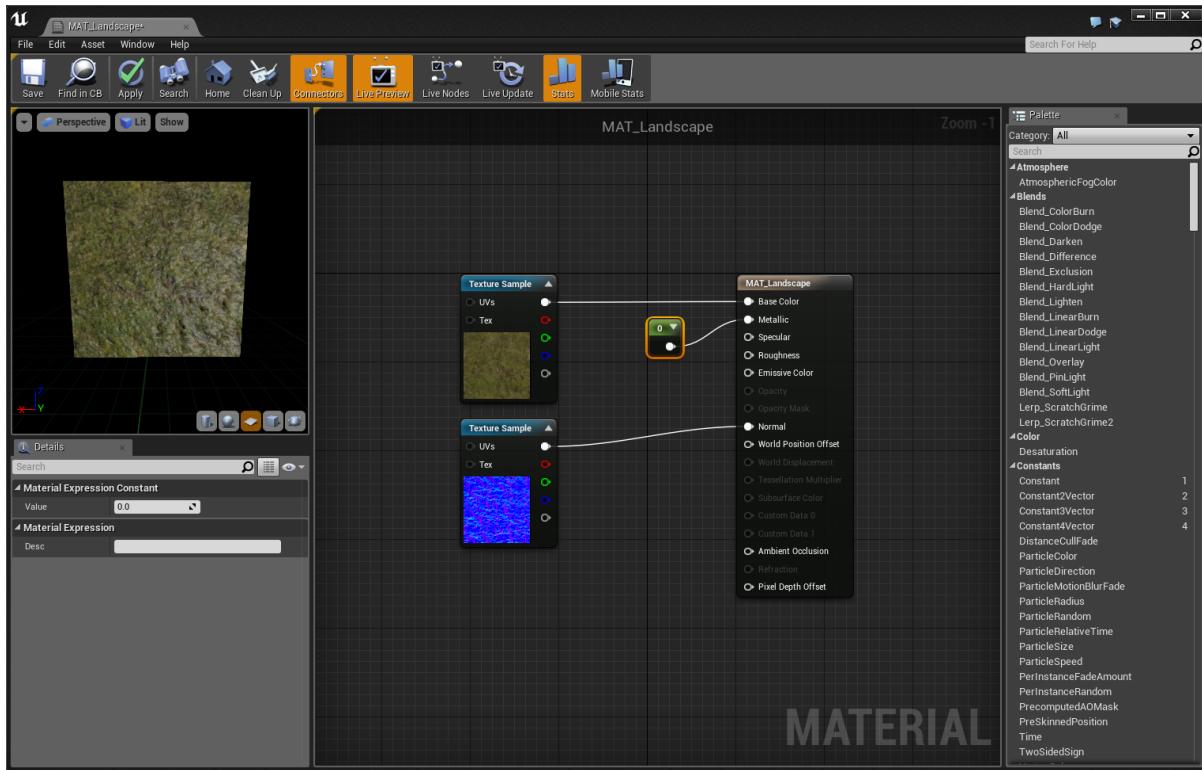


Figure 52

This can also be done to the ‘Roughness’ setting, the roughness setting determines how light is distributed, a setting of 0 makes the material reflect too much light, making it appear sparkly.

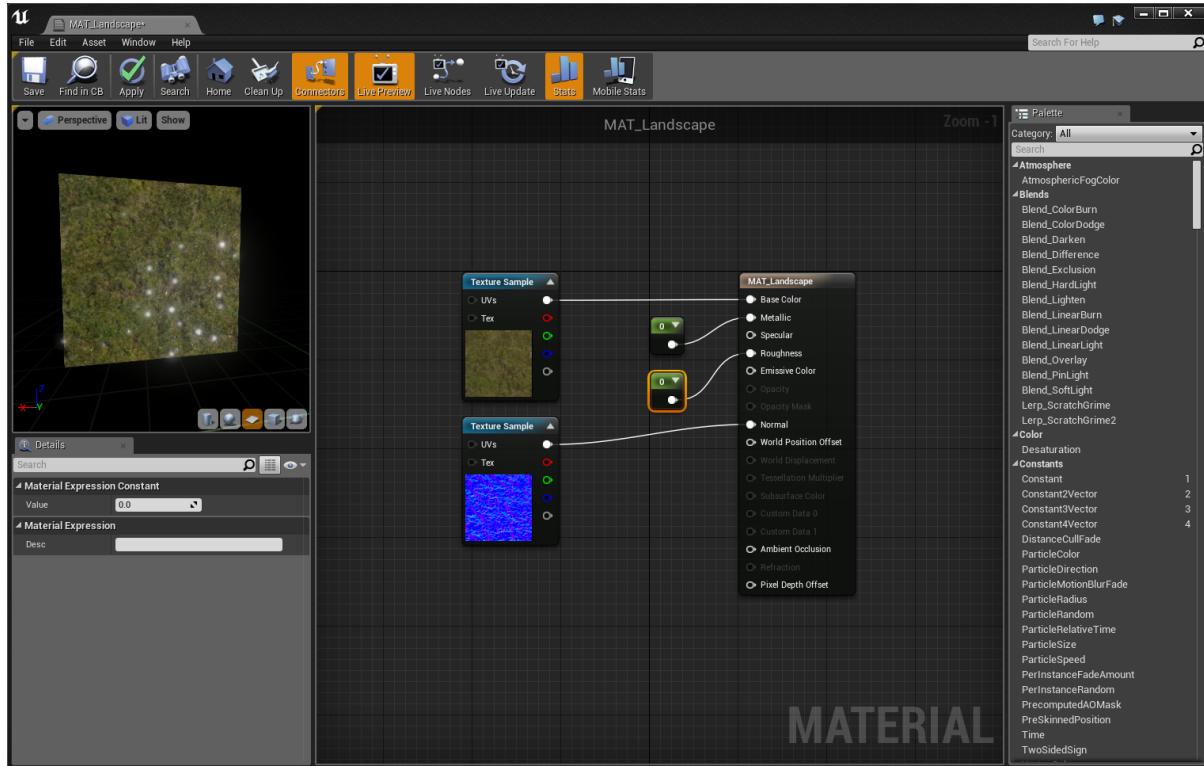


Figure 53

A setting a 0.8 seems to be a good value for this texture, it now looks very realistic.

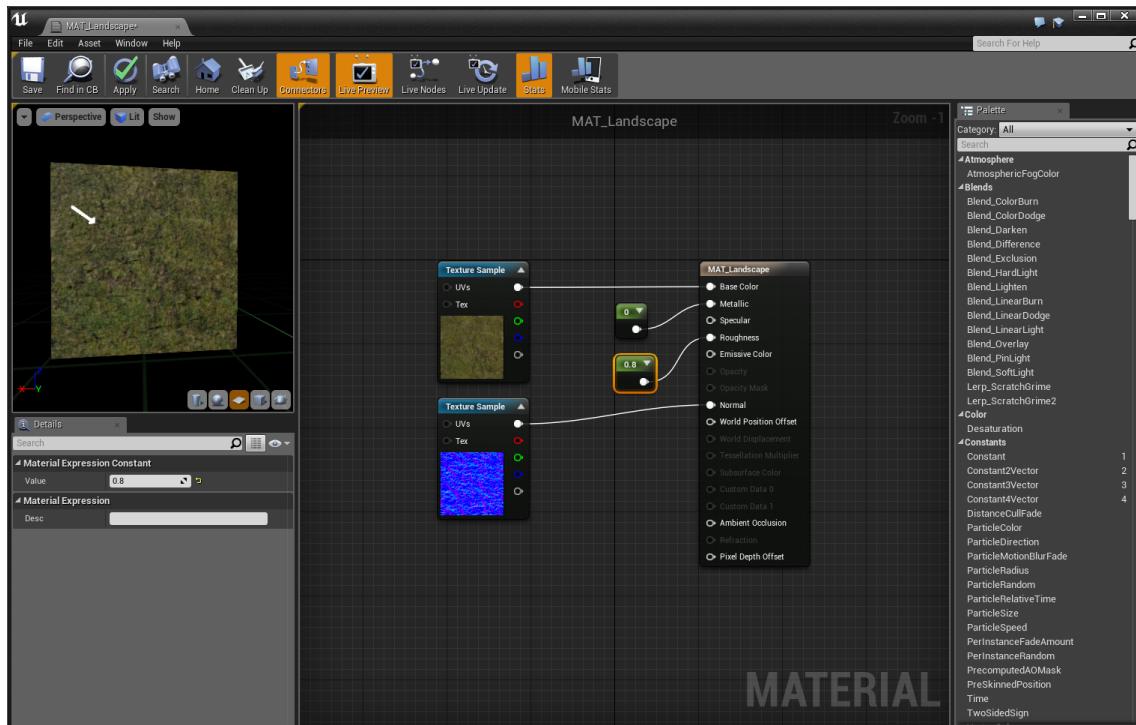


Figure 54

If this material were to be assigned to the landscape, nothing would happen, this is because the textures don't have any UV co-ordinates. A 'LandscapeCoords' has been added, and the white channel plugged into the 'UVs' slot for the texture and normal map.

The LandscapeCoords controls the UV co-ordinates for the textures, with this the scale, rotation etc. of the texture can be modified.

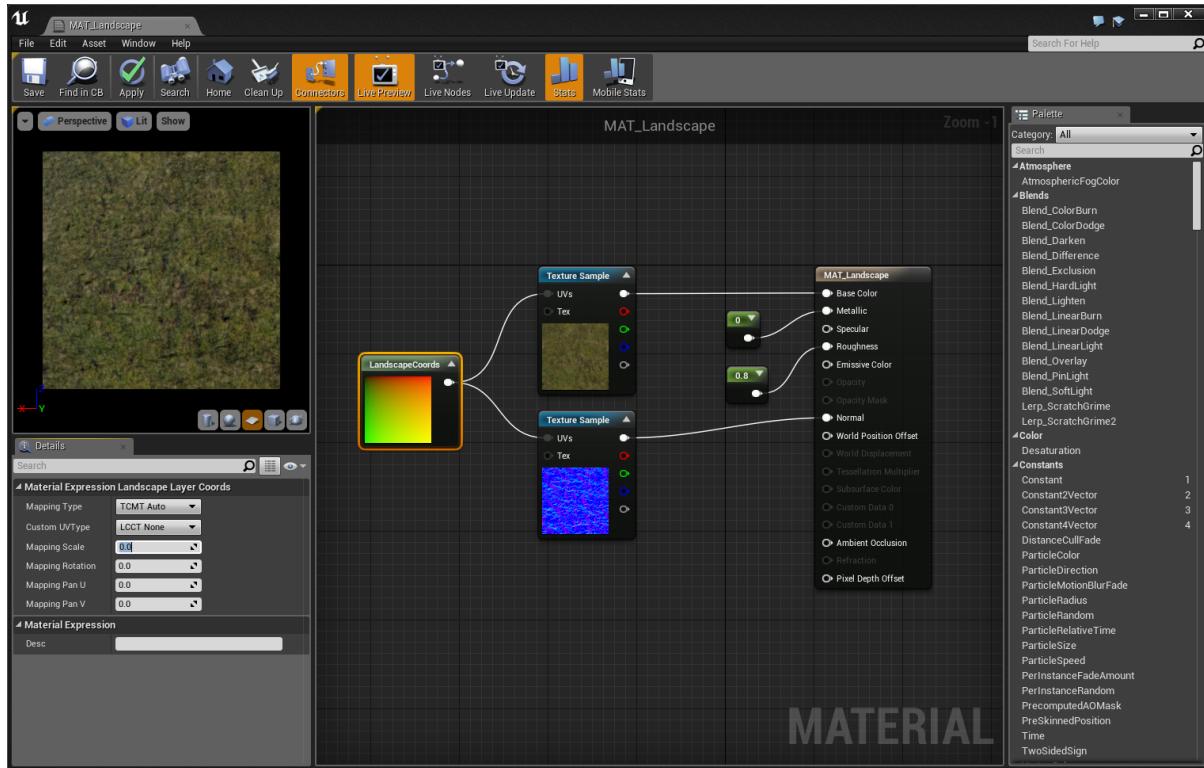


Figure 55

As an extreme example, the scale has been changed to 5, scaling up the texture five times, changing how it is displayed on the landscape.

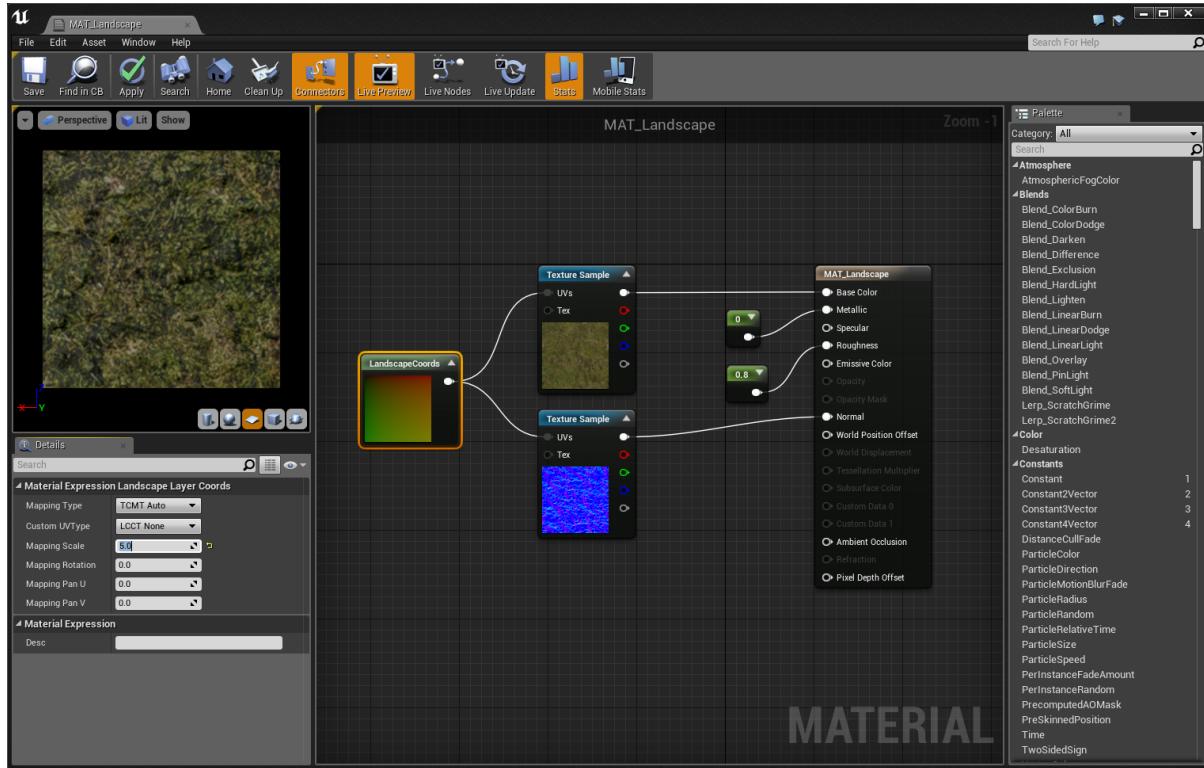


Figure 56

This material will have multiple textures, to ensure they all blend together nicely, a ‘LandscapeLayerBlend’ must be created.

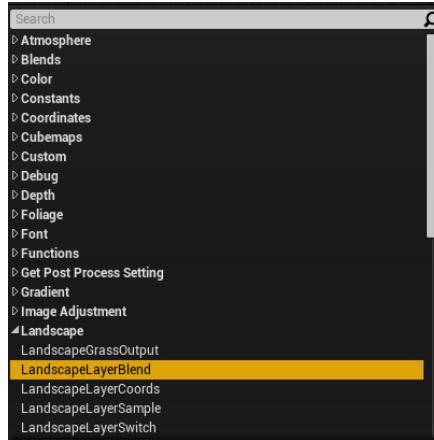


Figure 57 - Adding New LandscapeLayerBlend

Here the connections from the texture and normal map to the material have been removed, and the layer blend has taken their place.

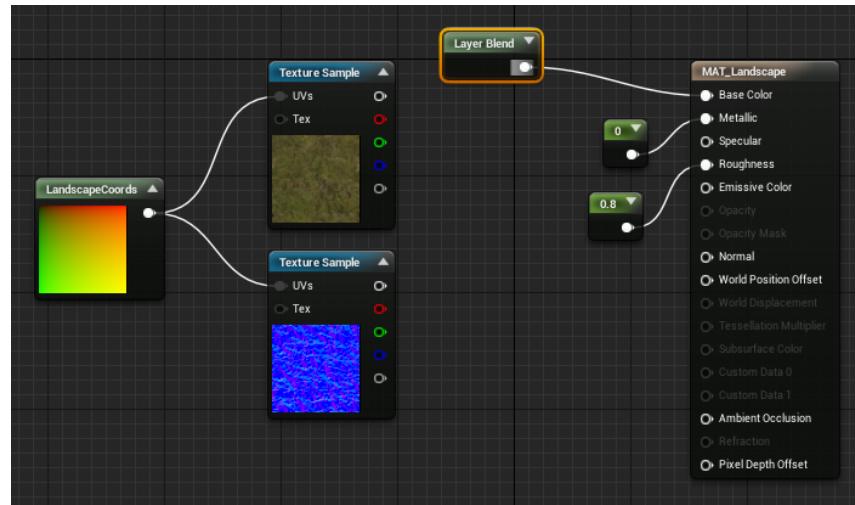


Figure 58

Three layers have been added to the layer blend, one for each texture that is to be added to the material.



Figure 59

The layers are currently being blended by weight, but for this material they need to be blended by height. This allows the textures to be nicely blended together.

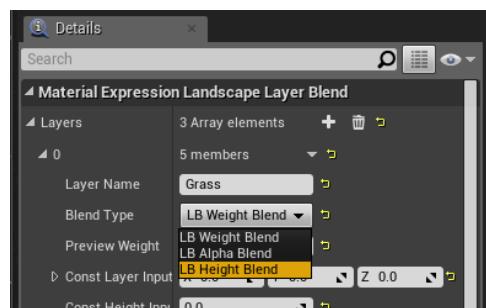


Figure 60

Here you can see new layers have been added after changing this option.

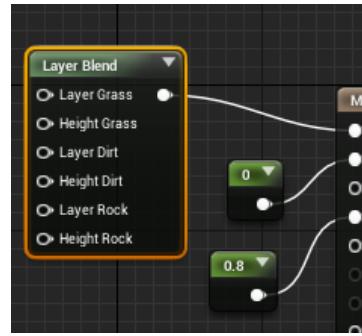


Figure 61

To ensure the layers blend correctly, the 'Preview Weight' must be changed. The preview weight works like layer system in Photoshop; stacked like paper, being able to see layers below through transparent parts of the upper layers. The preview weight for grass is set to 1 to make it appear on top, dirt is set to 0.5 to make it the middle, and rock set to 0 to make it the bottom.

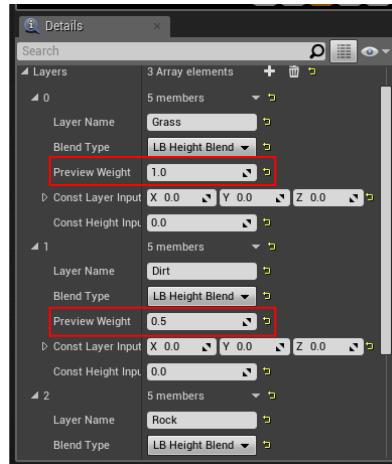


Figure 62

The grass texture can now be assigned to the layer blend, connecting it to the 'Layer Grass'

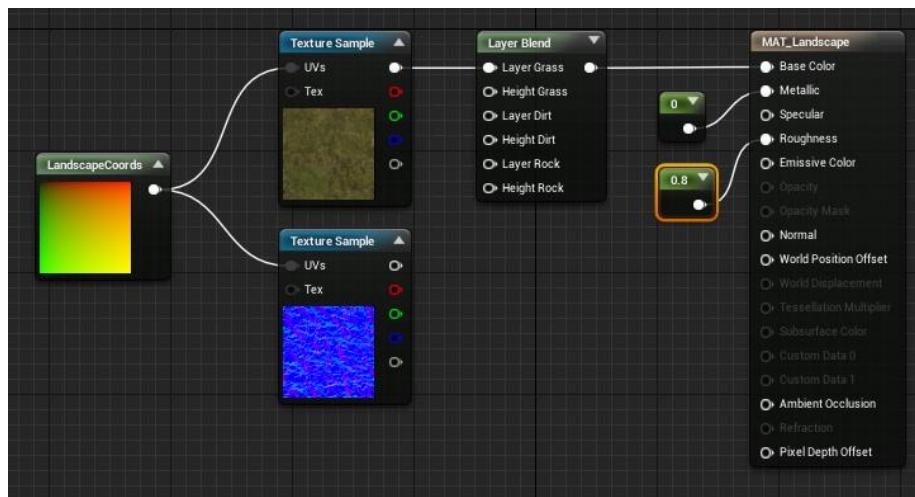


Figure 63

The layer blend needs a way of blending the different texture layers together. The alpha channel of the texture is used for this, this way they are blended using transparency, so they fade in to each other.

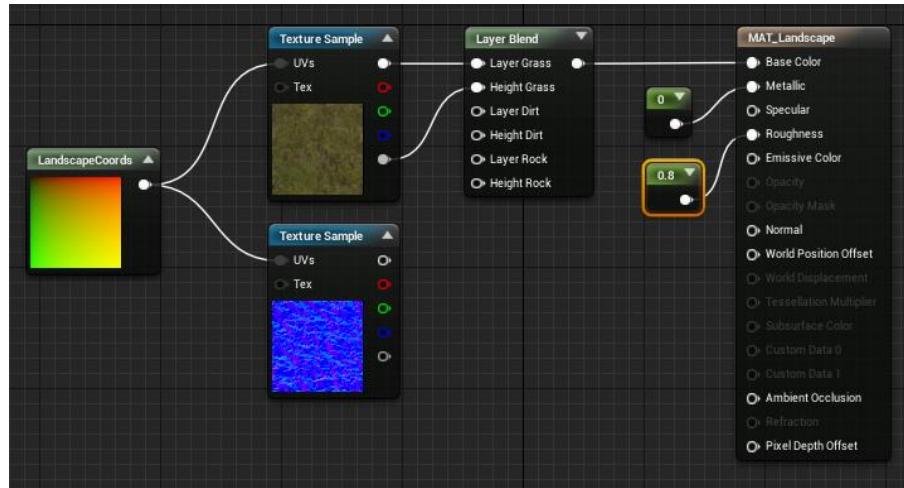


Figure 64

Below you can see that all the other textures have been added, and all of them plugged into the appropriate channels of the layer blend in the same way the grass texture was. The LandscapeCoords has also been plugged into all of the textures.

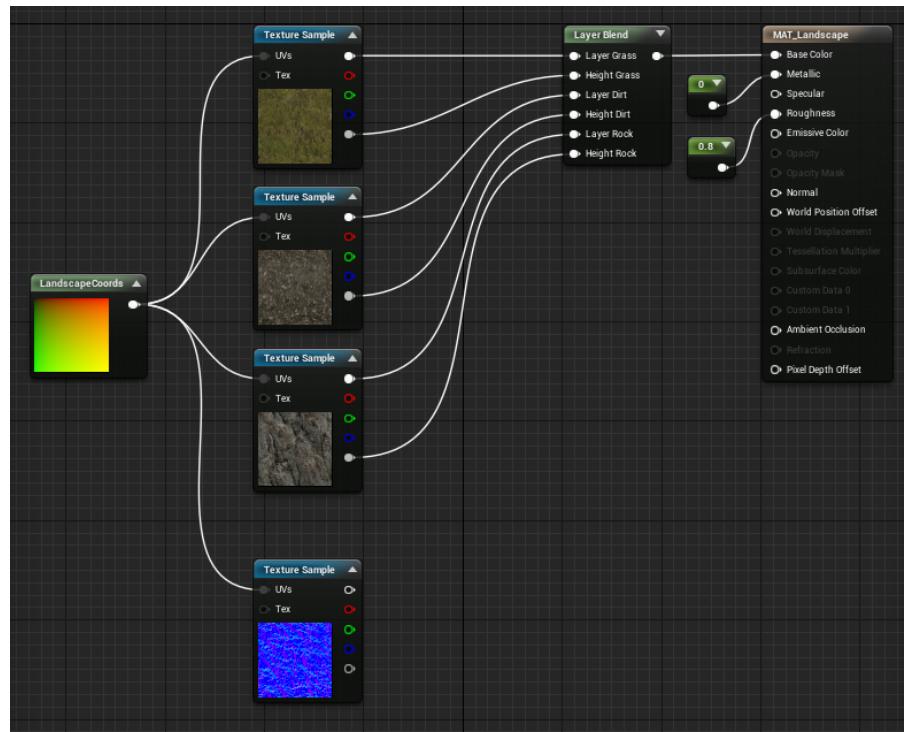


Figure 65

Below, the layer blend has been duplicated to be used in the same way as the first one, but for the normal maps. All the normal maps have been connected in the same way as the textures just were.

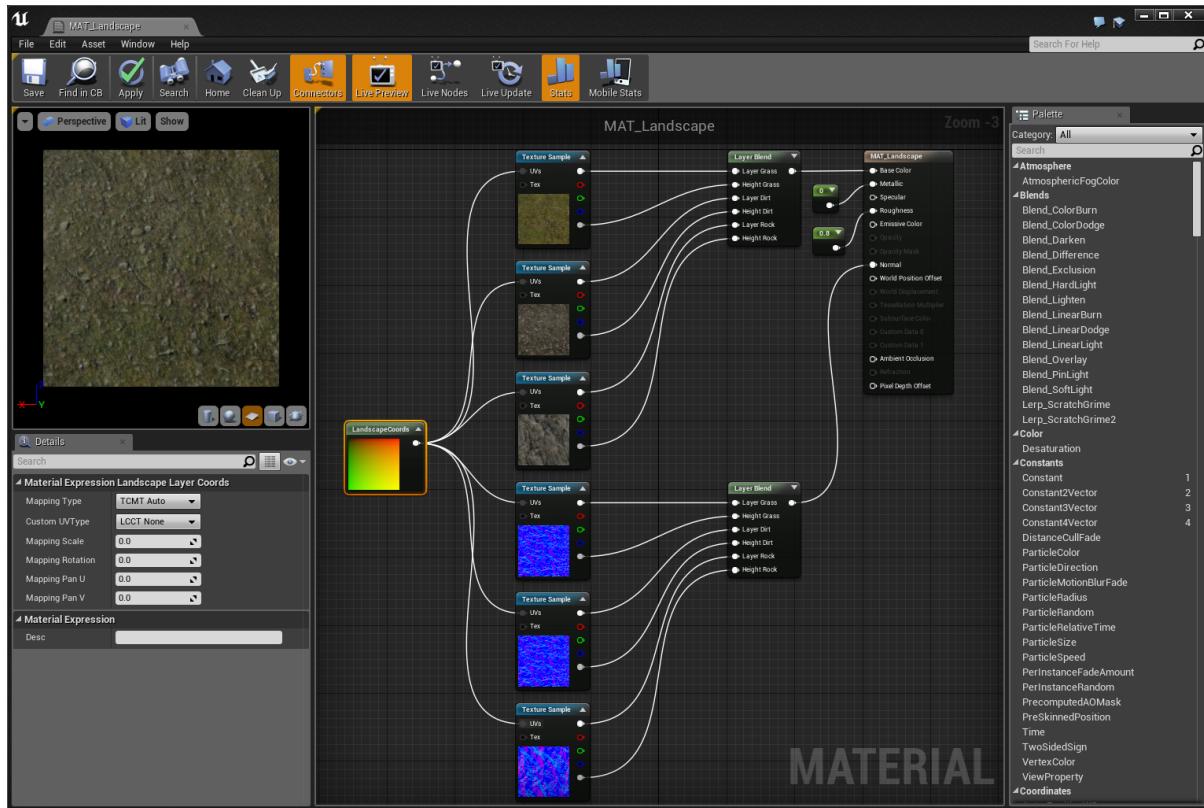


Figure 66

To neaten things up, comment boxes were made around each group of objects. This has been called “Diffuse Textures”

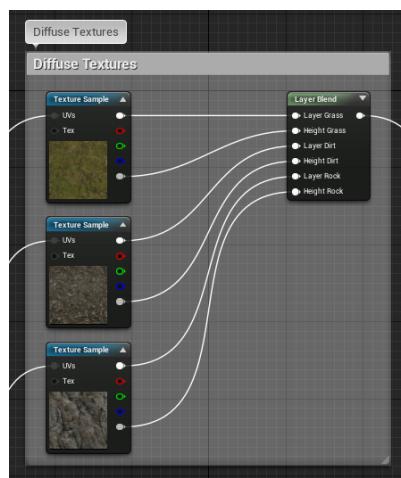


Figure 67

The normal maps have been labelled “Normal Maps”

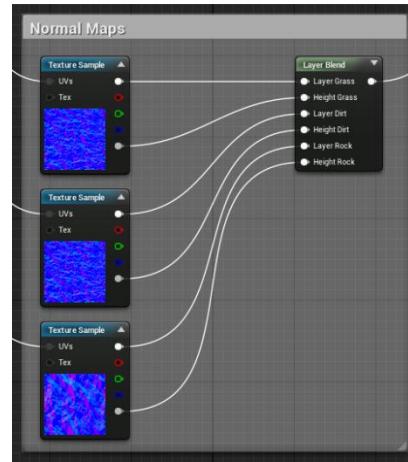


Figure 68

The LandscapeCoors has been labelled “Landscape UV”.



Figure 69

Landscape is black when material is assigned to the landscape, this is because the landscape has no layer information assigned to it, which is how Unreal knows how the textures blend together.

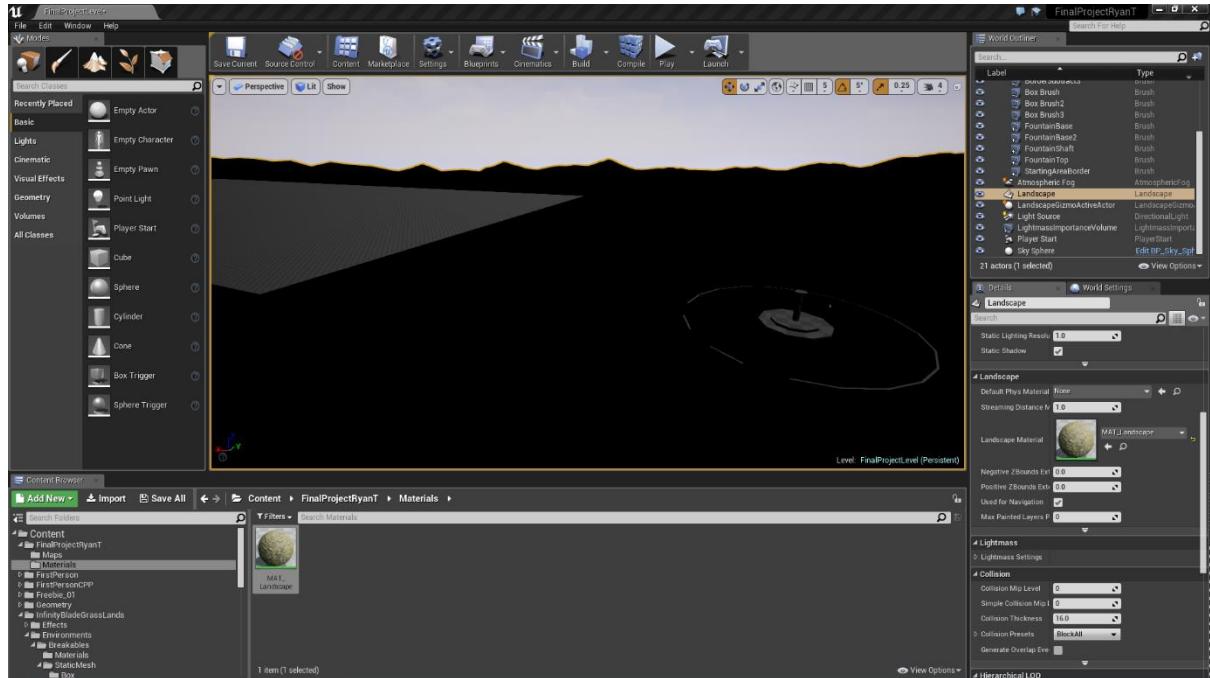


Figure 70

To solve this, new information layers must be created for each of the material layers. This is done by clicking the plus button and selecting ‘Weight0Blended Layer (normal)’.

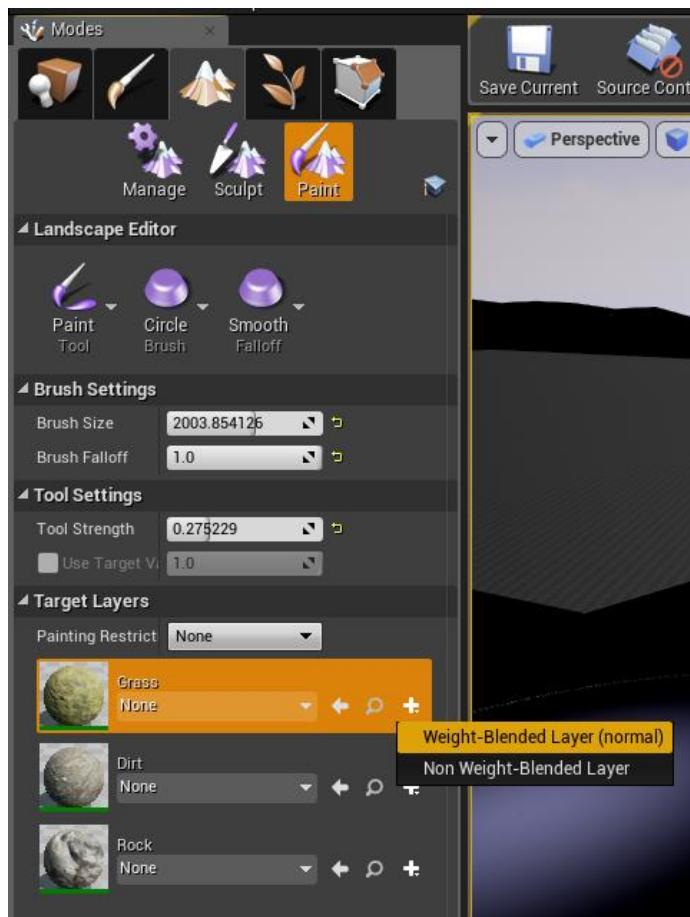


Figure 71

This is repeated for all three textures, and the resulting layers saved in the layers directory.

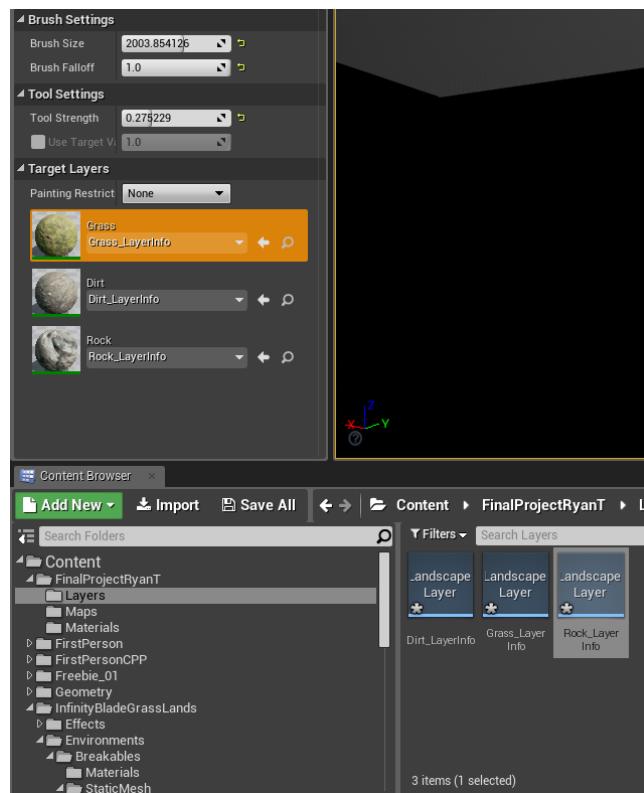


Figure 72

The scale of the texture is too small however, and appears tiled on the landscape, meaning the same texture can be seen multiple times over and over, like a tiled floor.

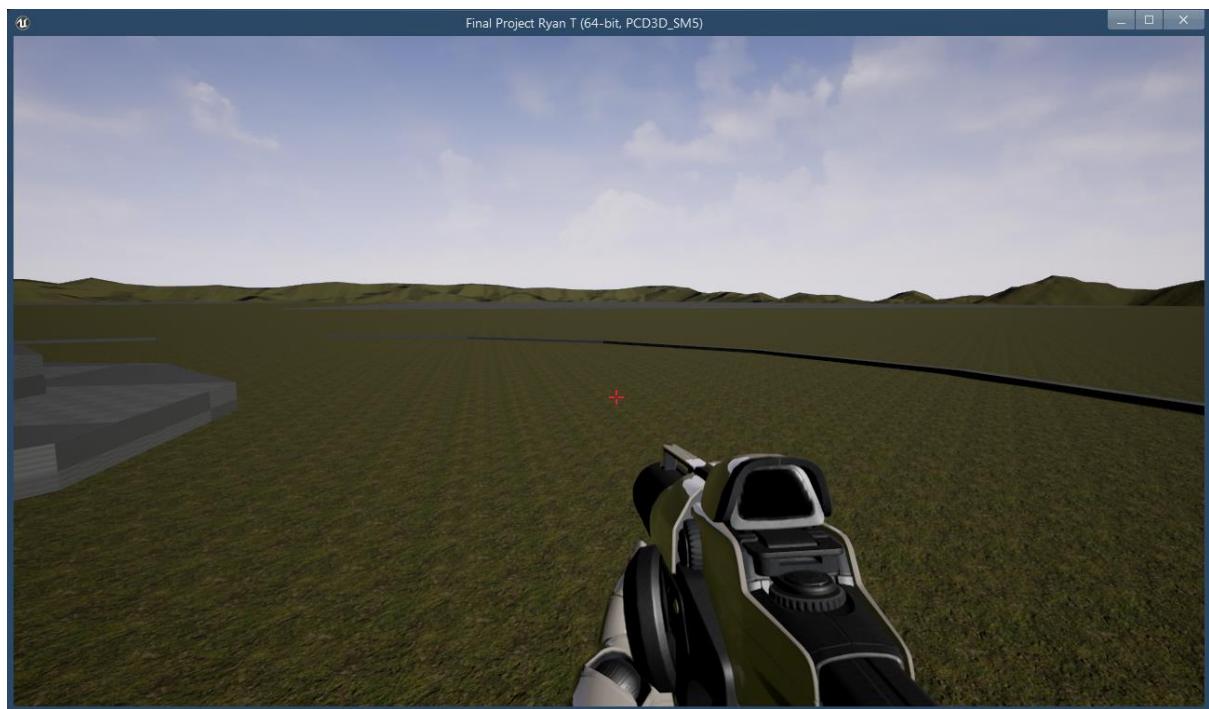


Figure 73 – Landscape Texture Tiling

This can be fixed by changing the scale of the material to increase the size, 4.5 in this case seems good.



Figure 74 - Changing Mapping Scale to 4.5

The tiling is now much less noticeable.

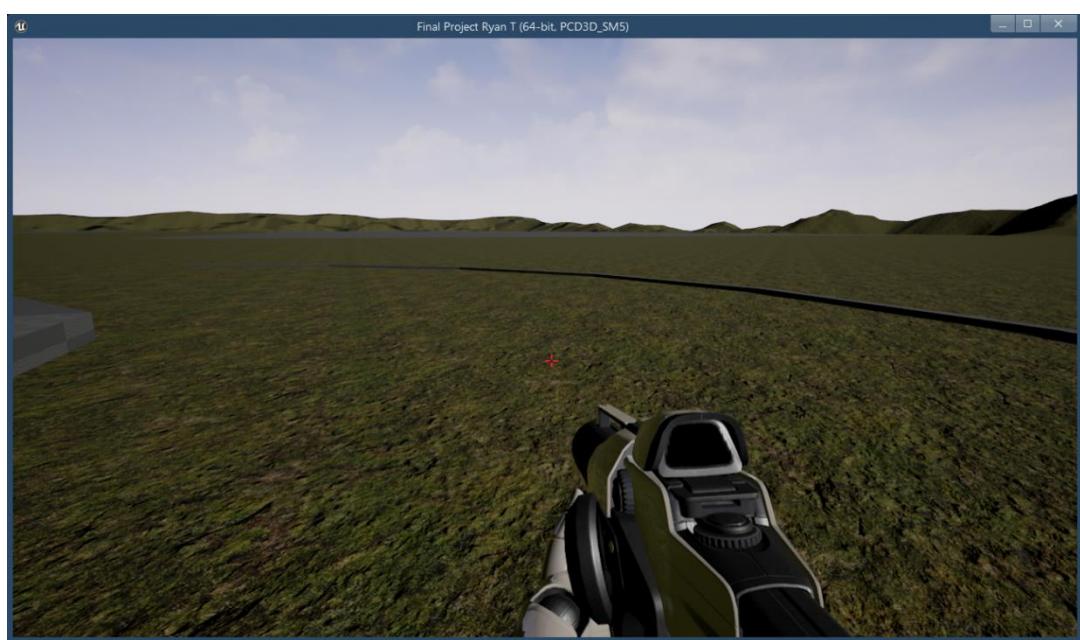


Figure 75 - Landscape Texture Tiling Fixed

Selecting another texture in the layers pane allows this new texture to be painted on top of the base grass texture.

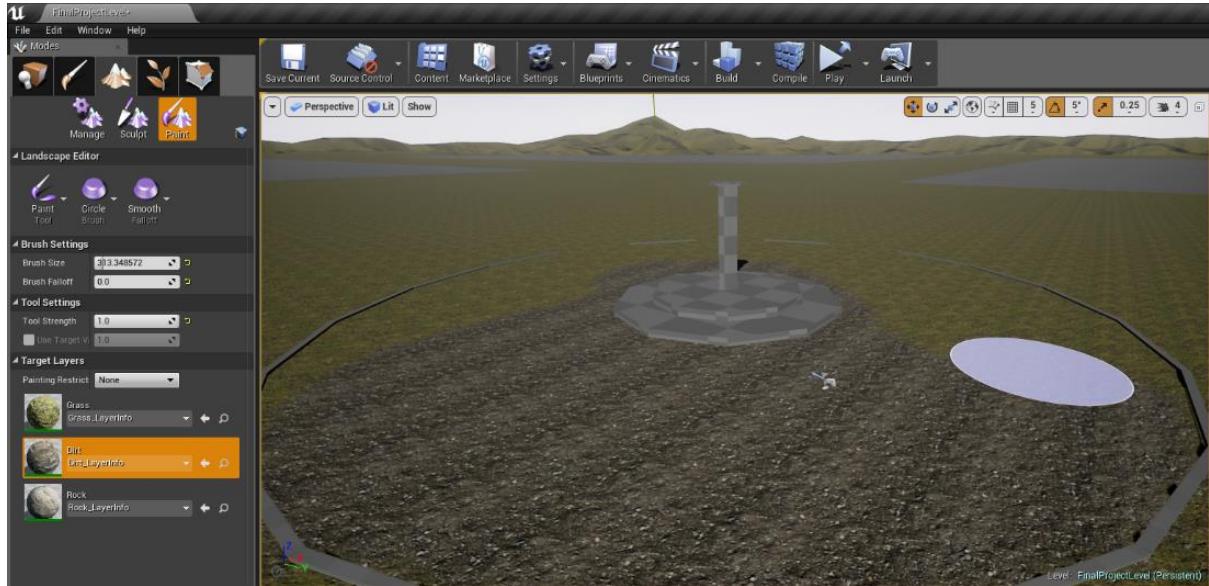


Figure 76 - Texturing Starting Zone

Below you can see the texturing for the starting area is complete.

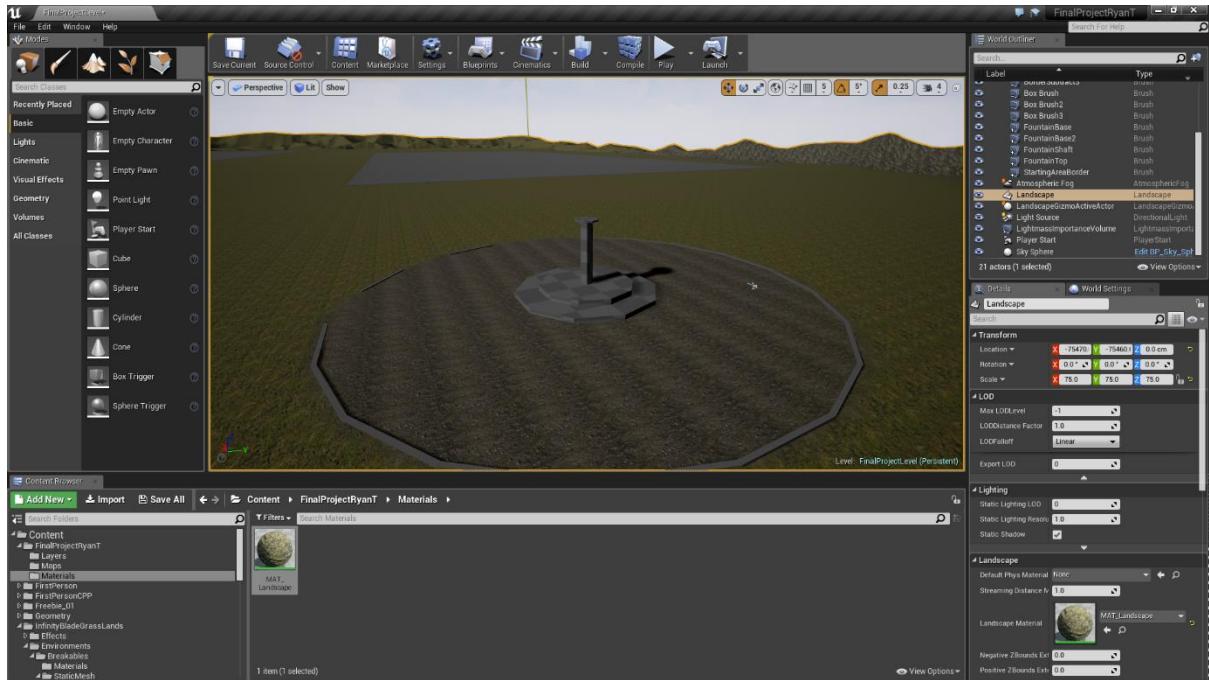


Figure 77 - Texturing Complete

Next the texturing for the mountains can begin. Using a low strength and high falloff for the brush settings allows for a nice, soft blending effect.

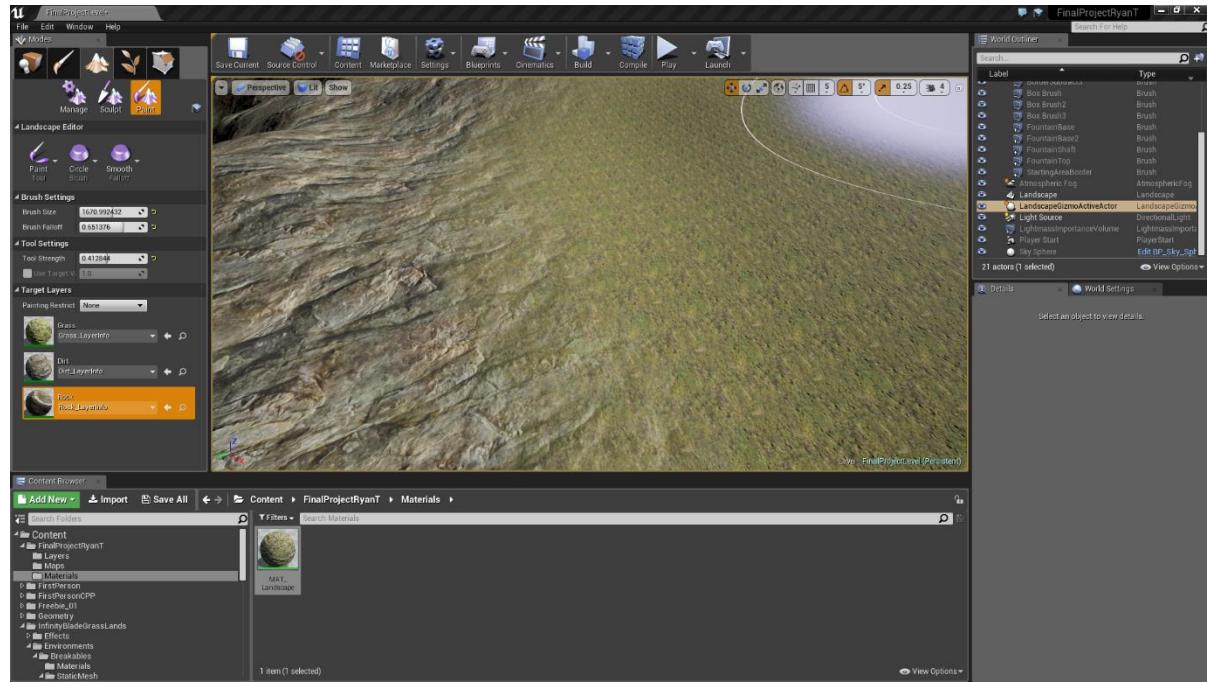


Figure 78 - Texture Blending

Using similar settings for the dirt texture creates a nice, smooth and seamless blend between the two.

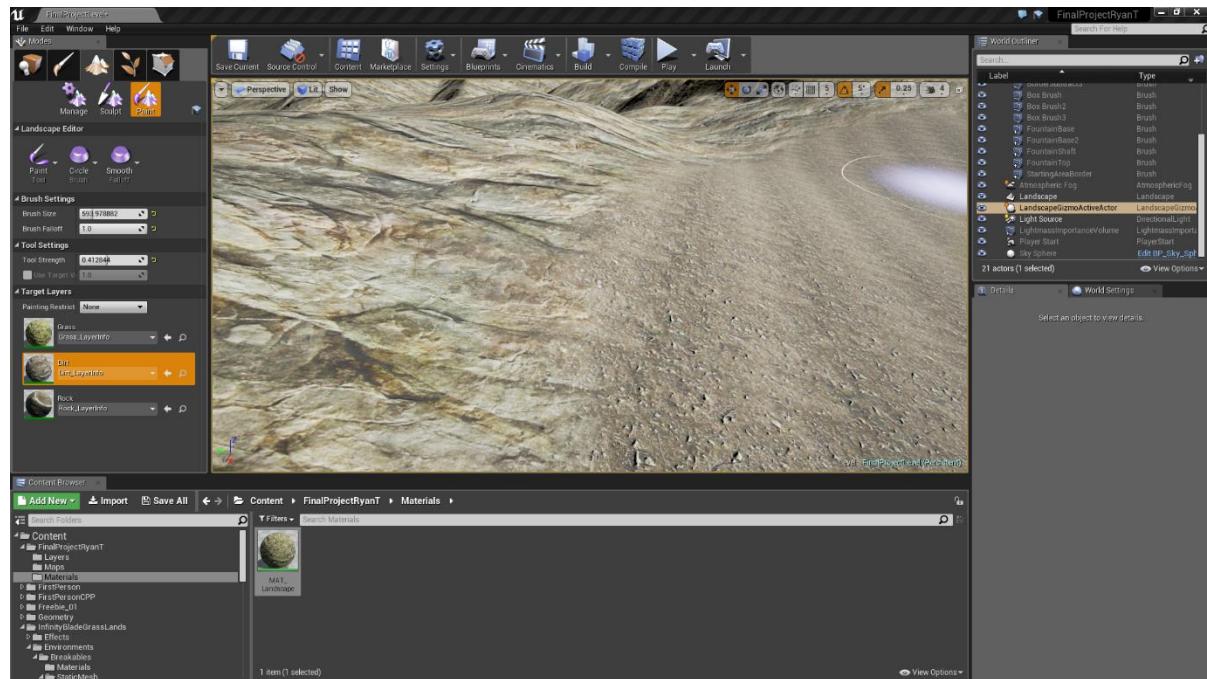


Figure 79 - Texture Blending

Painting the dirt texture with a very hard edge allows it to be painted accurately, it can then be blended into the surrounding textures with the smooth tool.

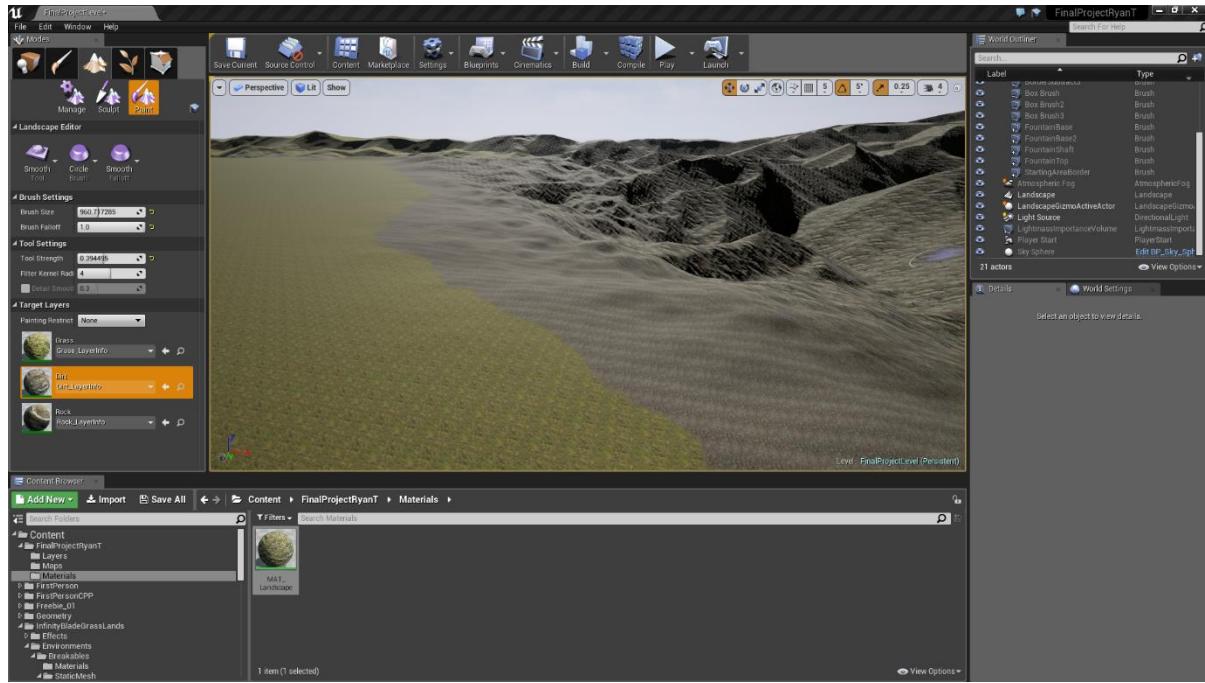


Figure 80

Below you can see the result of using the smooth tool.

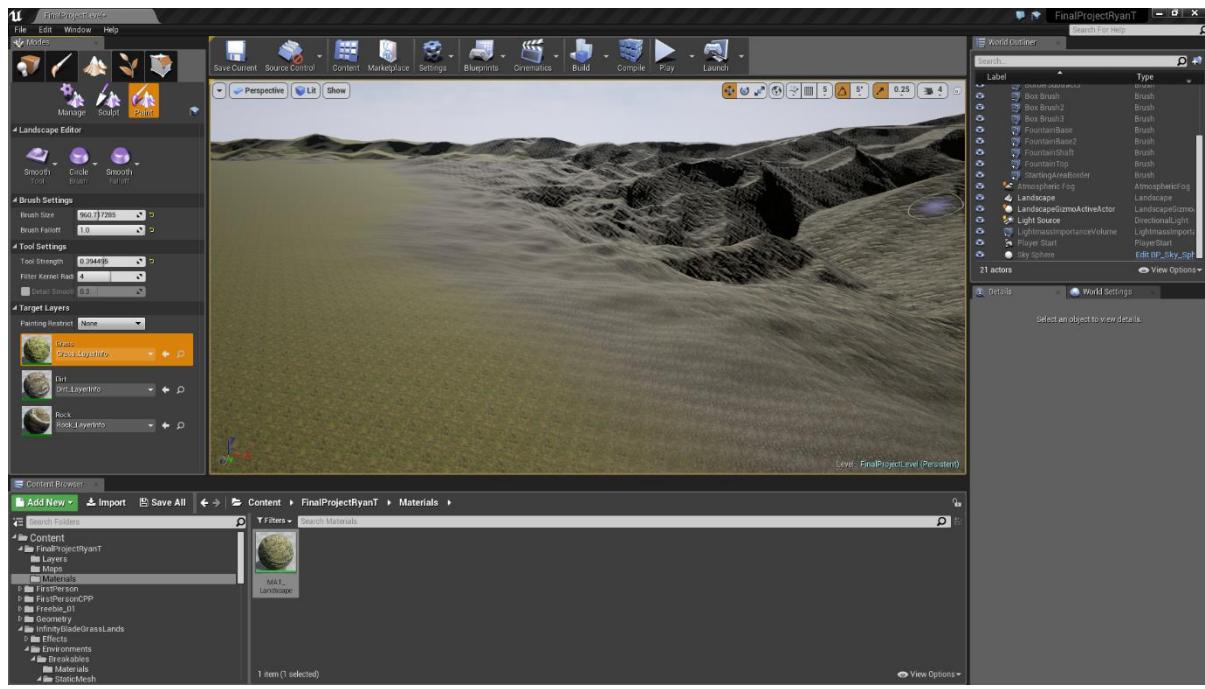


Figure 81 - Texture Blending

Below, a second grass texture has been added to improve landscape blending and give a greater variation in textures.

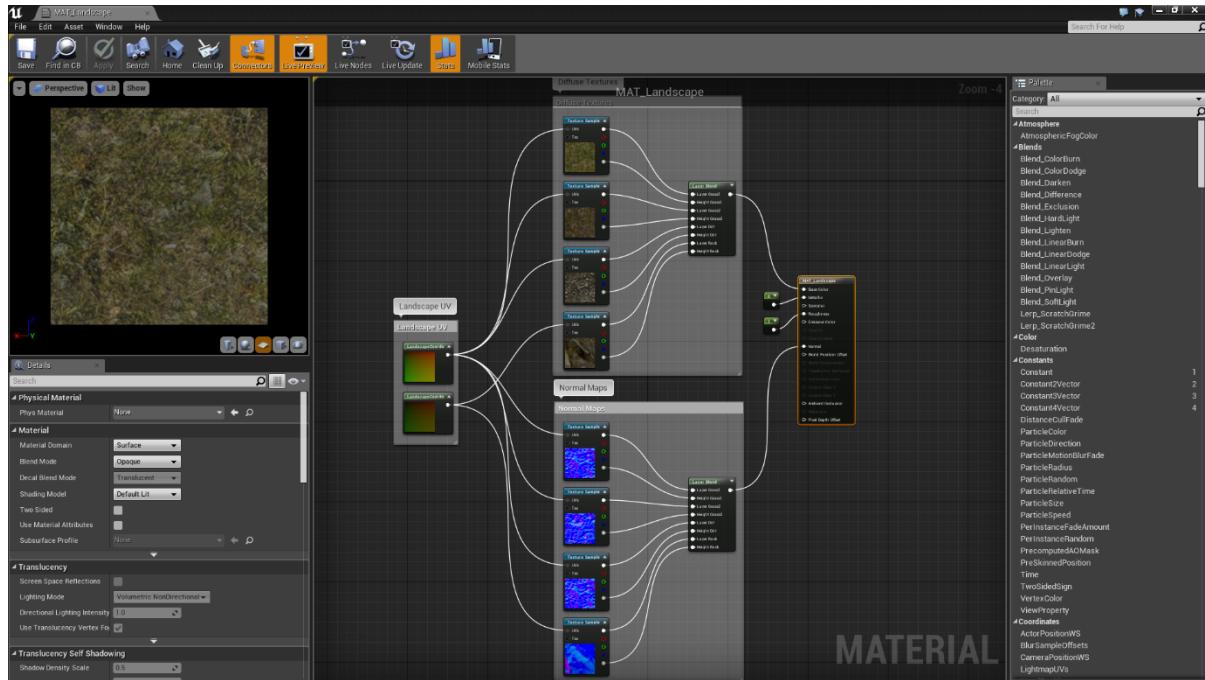


Figure 82

Top down view of finished landscape, here the blending of the four textures is visible.



Figure 83 - Overview of Texture Blending

In the figure below, you can see the the spline tool being selected. This tool is used for the creation of paths and roads.

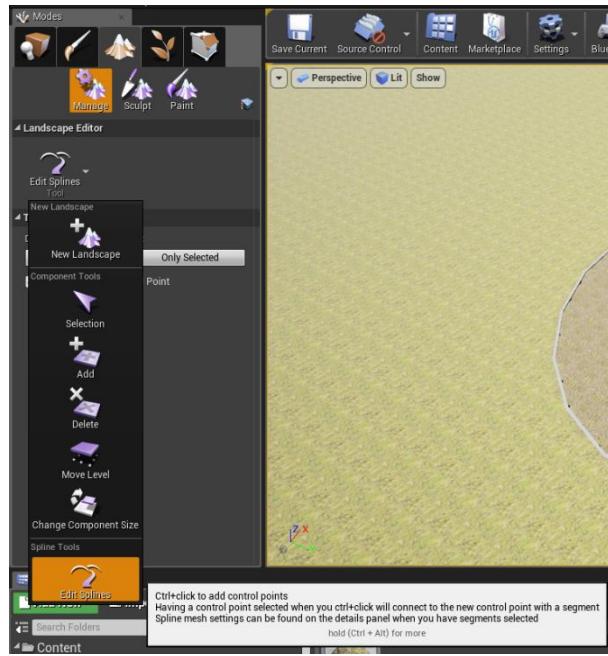


Figure 84

Below you can see the completed first path, made using the spline tool.

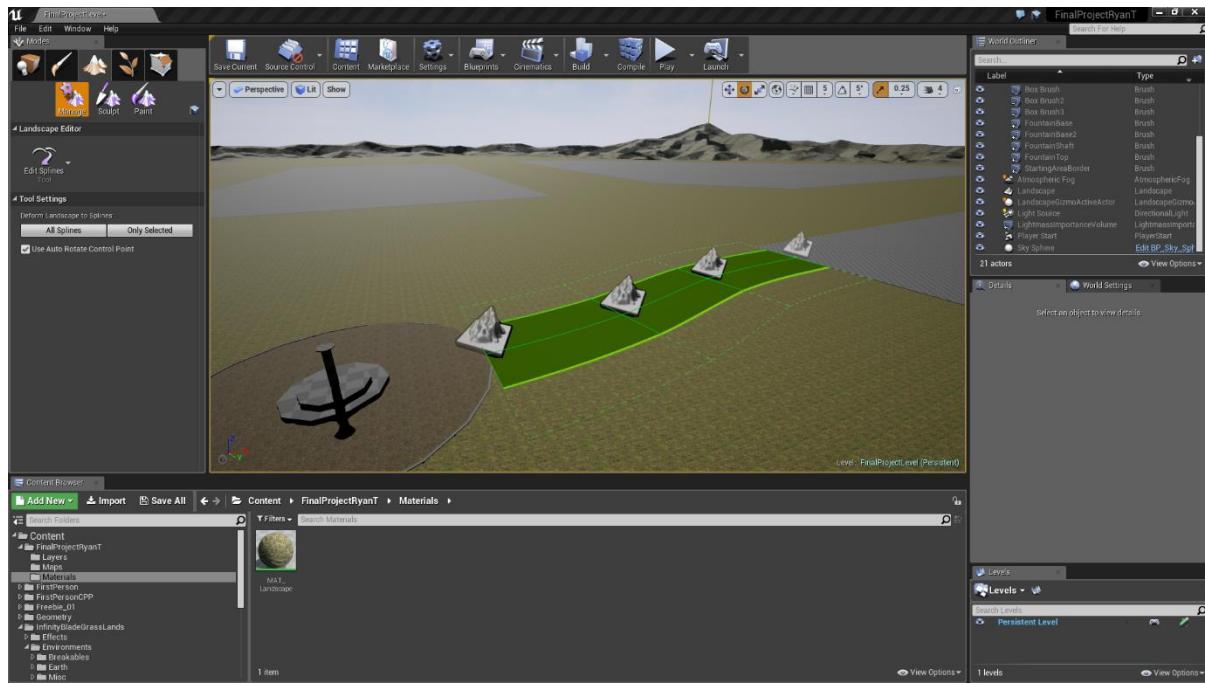


Figure 85 – Path Created with Spline Tool

The path is too wide, to fix this the 'Width' property has been changed from 1000 units to 320.

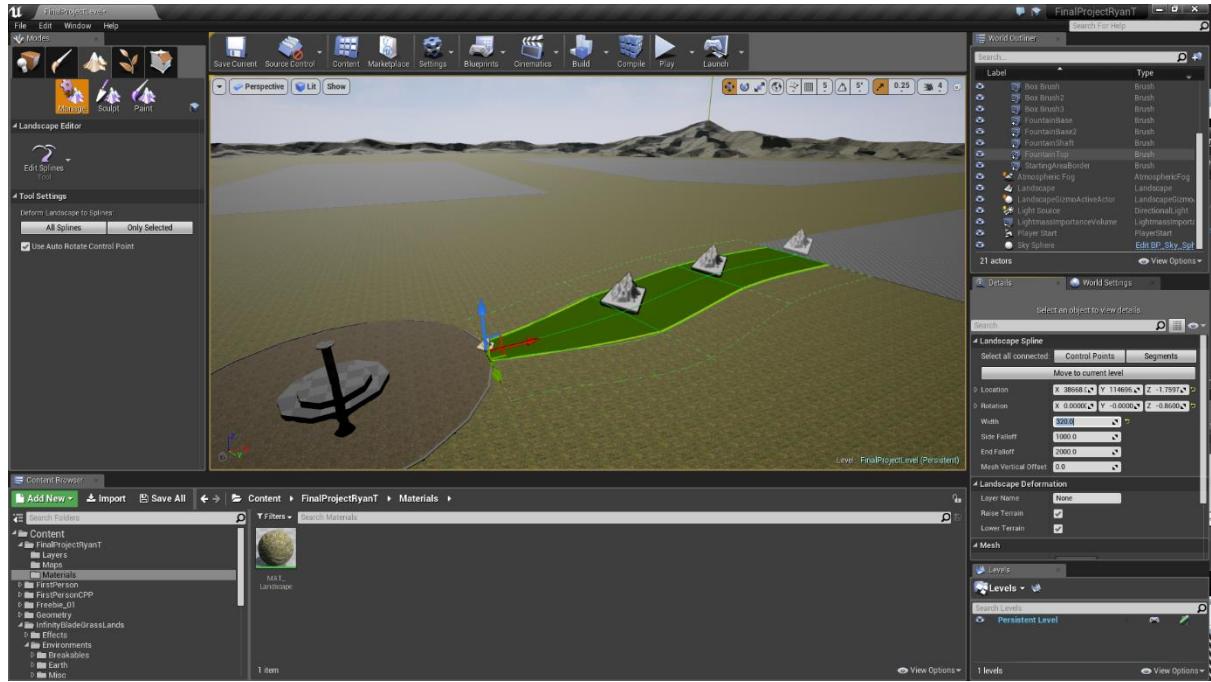


Figure 86 - Changing Path Width

The path is now the correct size, to break up the landscape one of the splines has been moved below the landscape, and one moved up. This will make the path uneven.

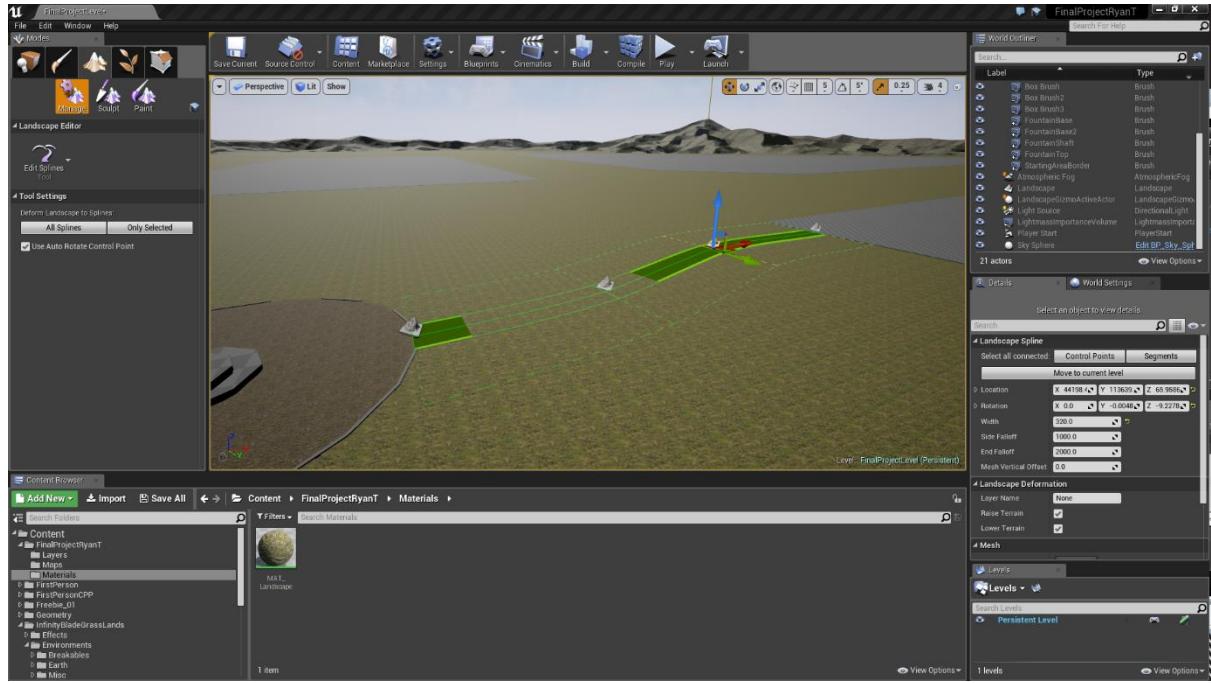


Figure 87

The landscape has been deformed to the path, a dip has been made where the spline was under the floor, and a hill where it was above it.



Figure 88

Below you can see all three paths completed, and the landscape deformed around them.

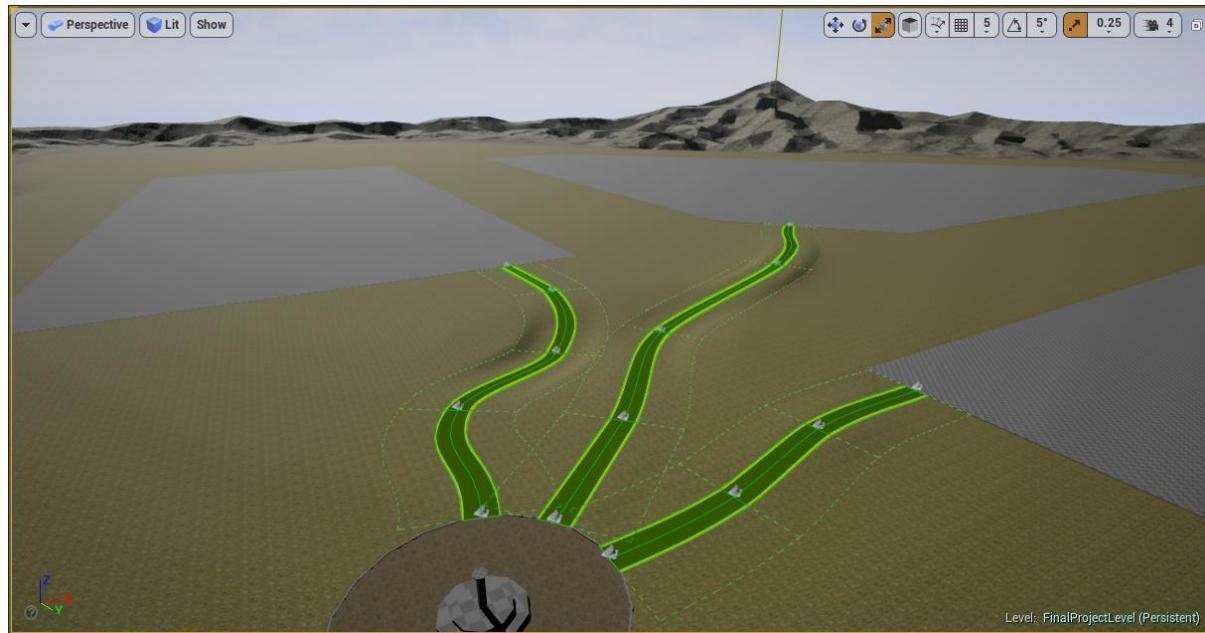


Figure 89

In the figure below, a mesh has been added to the path.

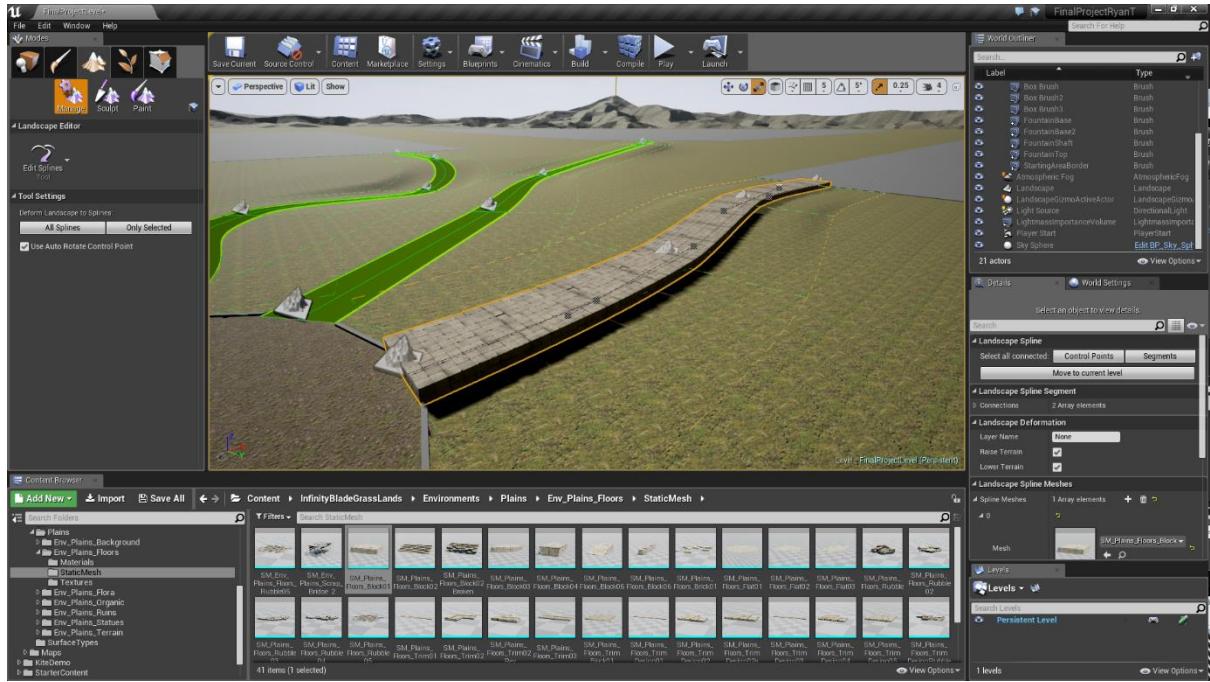


Figure 90 - Mesh Added to Path

The path was too high up so it was moved down until it is almost level with the landscape

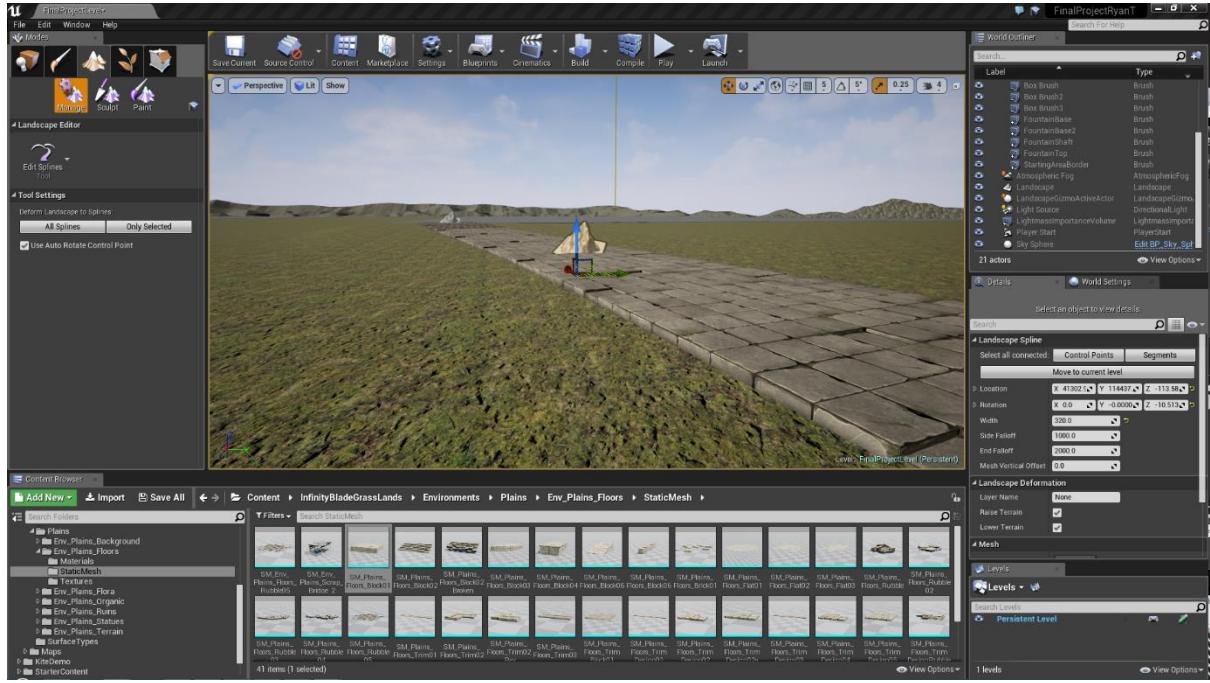


Figure 91

In the figure below, you can see the three completed paths.

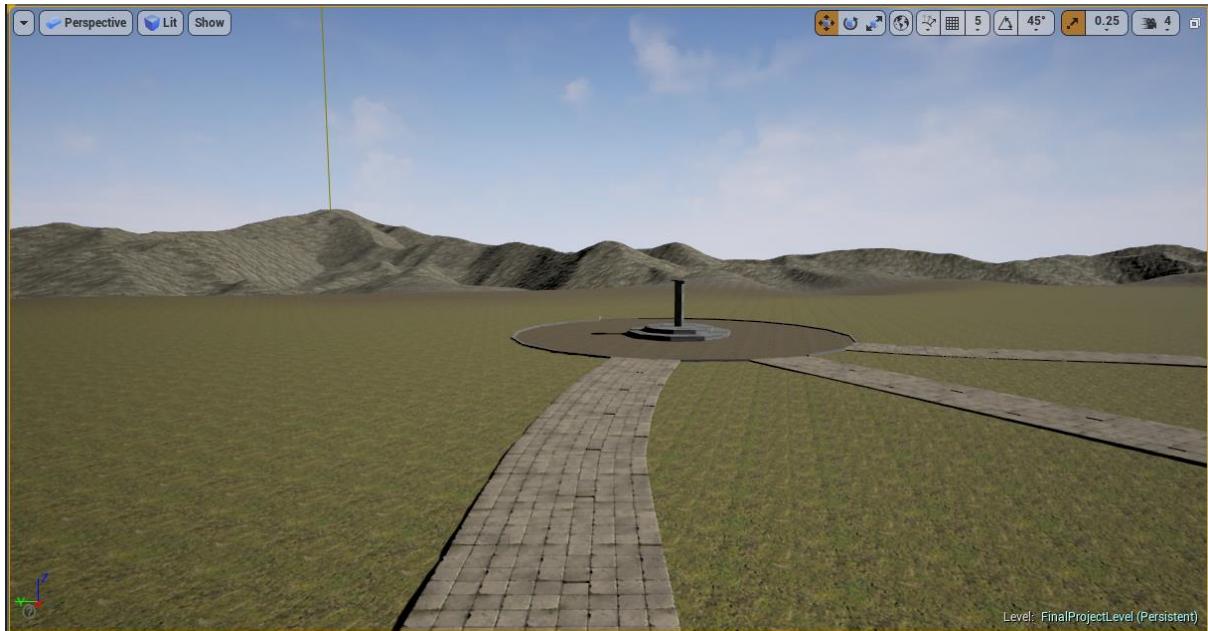


Figure 92 - Completed Starting Zone with Paths

Creating new sub-levels to split three areas up.

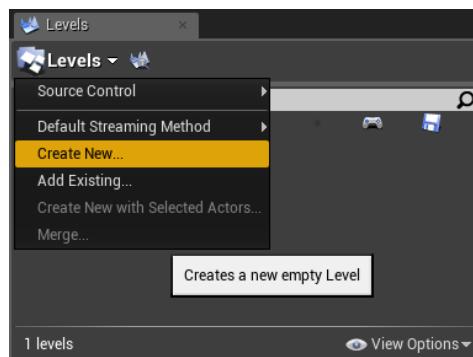


Figure 93 - Sub-Level Creation

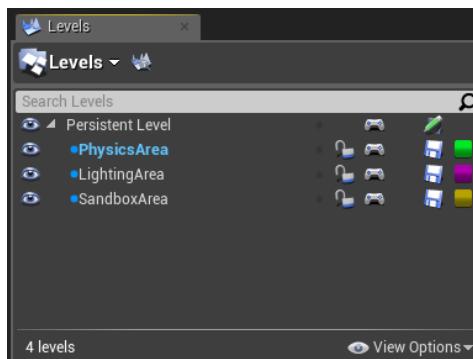


Figure 94 - Three Created Sub-Levels

The physics area can now be moved into its sub-level.

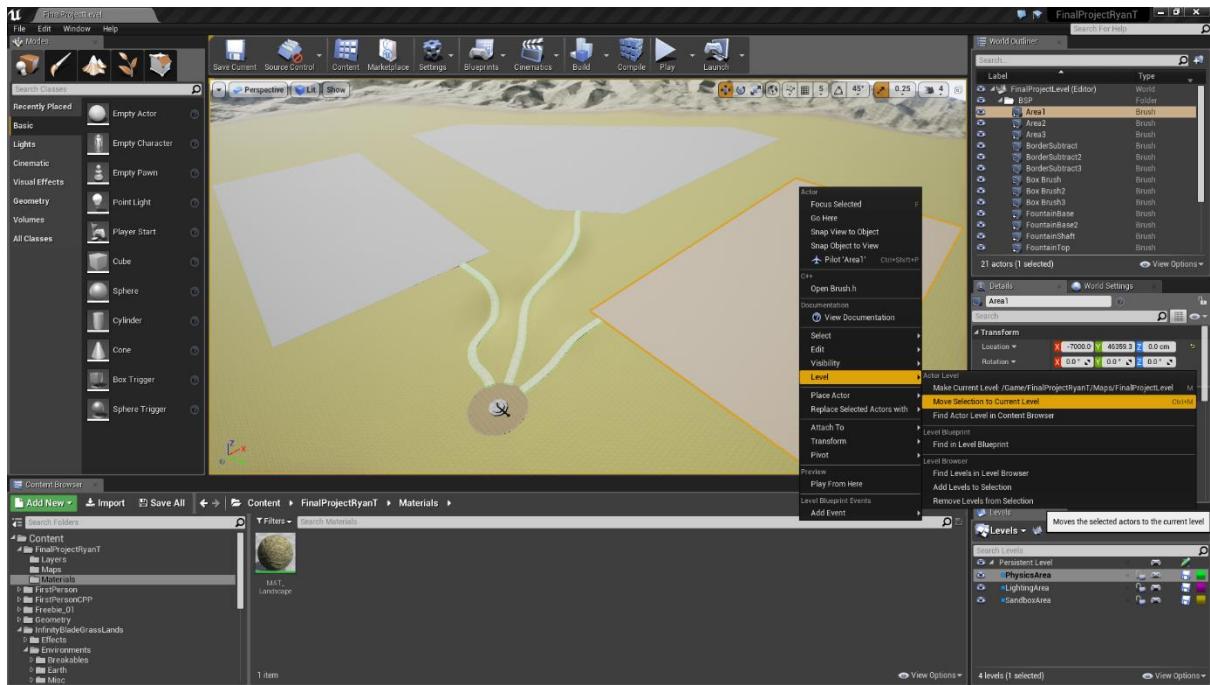


Figure 95

Streaming method set to blueprint, this means that the sub-levels will only load when a blueprint tells them to, useful for large maps that can't load full level.

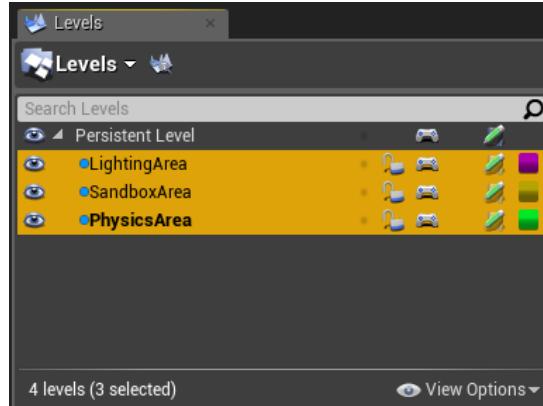


Figure 96

For now however, all levels will be set to always load, so everything will load in the future, level loading can be handled with a blueprint.

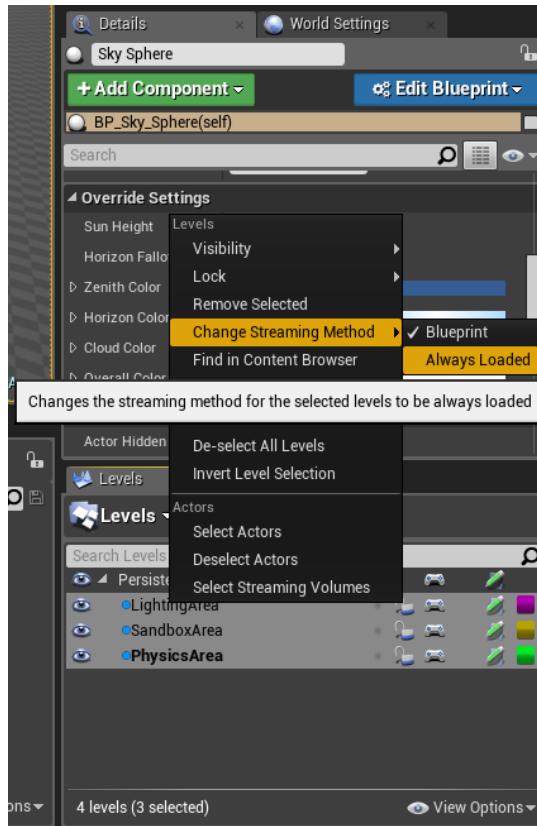


Figure 97

## 6.2. Collision Detection Building (Physics Area)

This section details the creation of the collision detection building and demonstration.

Using BSP tools, a box is being placed in the physics area. This will be the placeholder for the first building in the area.

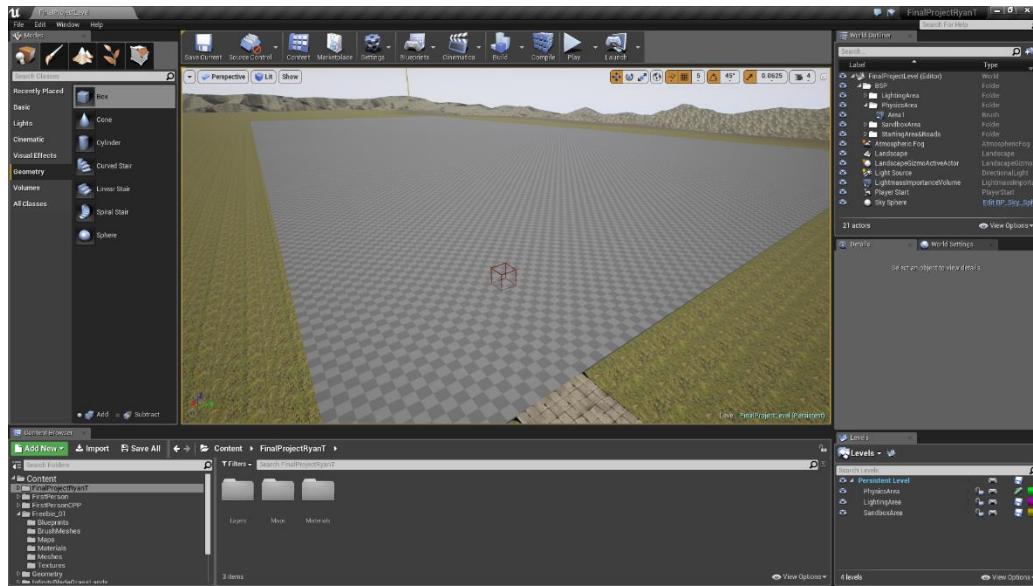


Figure 98

The size and shape of the box has been changed to match the size of the final building. This will make adding the final building a simple very easy. The box has been hollowed out by clicking hollow to change cube into room.

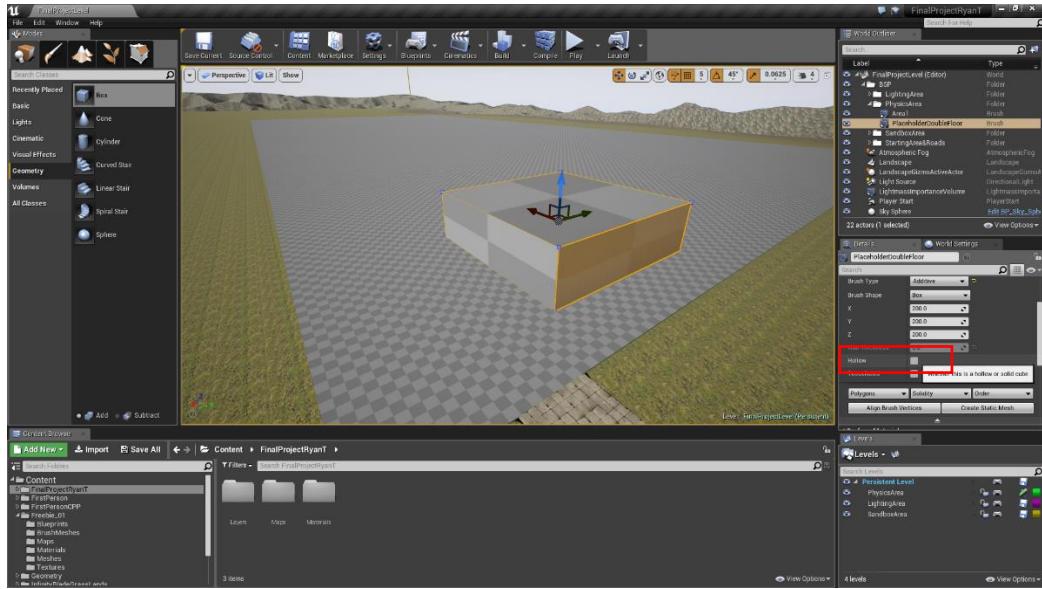


Figure 99

Here a box is being added, it will use the subtraction option displayed earlier in this document to cut a hole in the building to make a door.

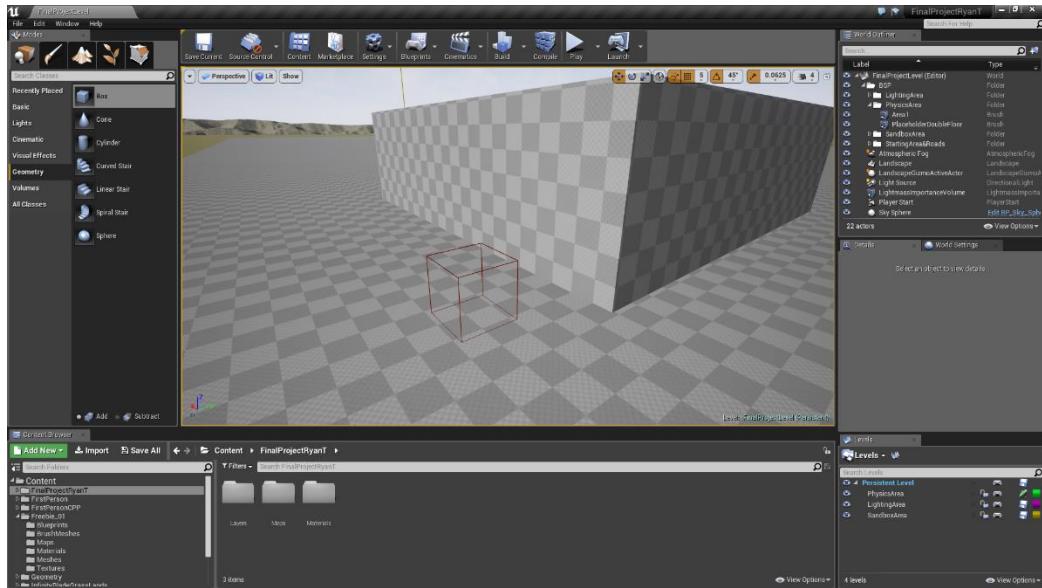


Figure 100

The box has been set to subtraction mode, and moved in place to create a doorway in the building.

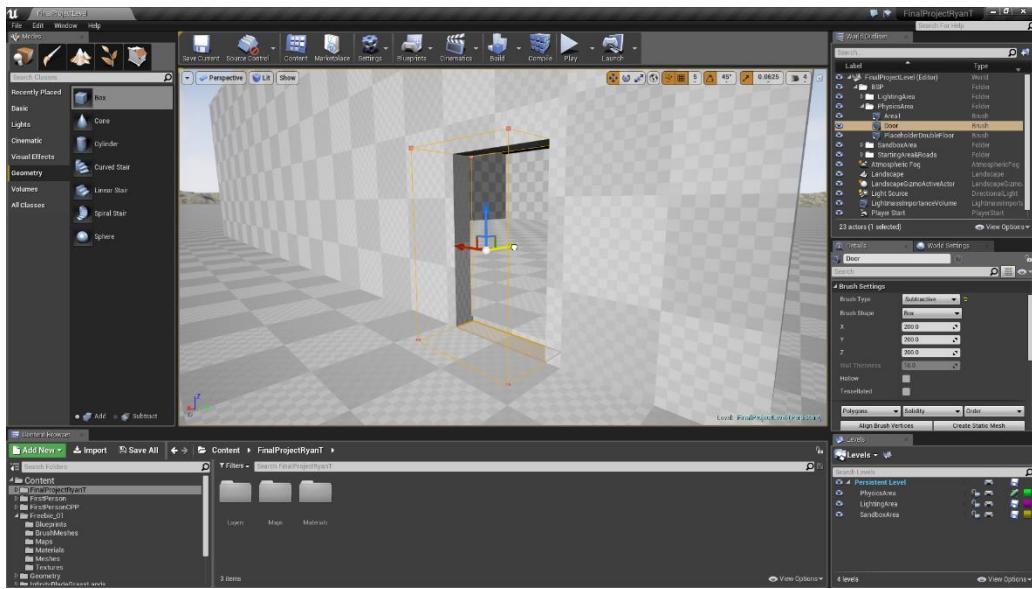


Figure 101

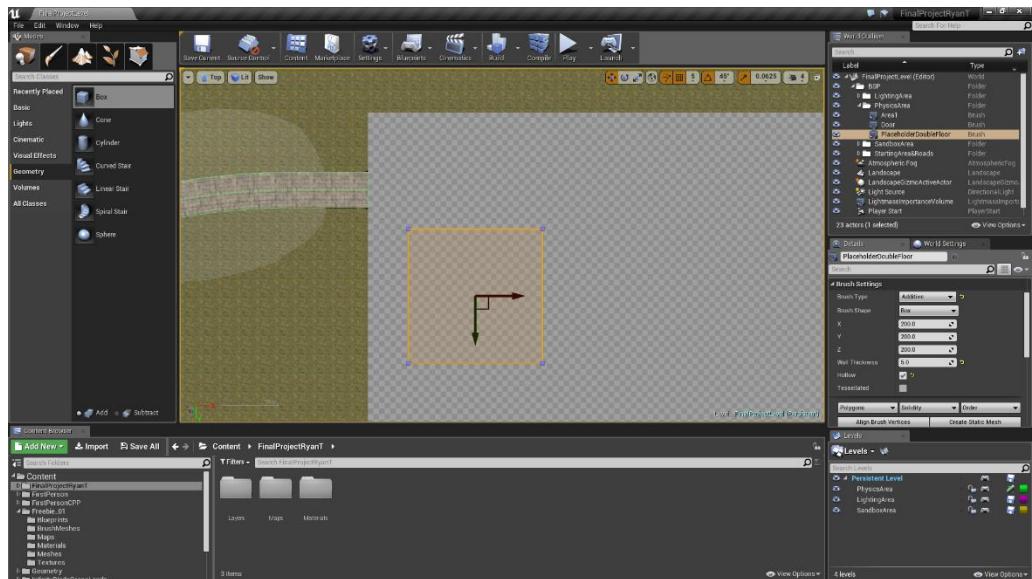


Figure 102 - Position of Building in Physics Area

Windows have been added to the building using the same tools as the door.

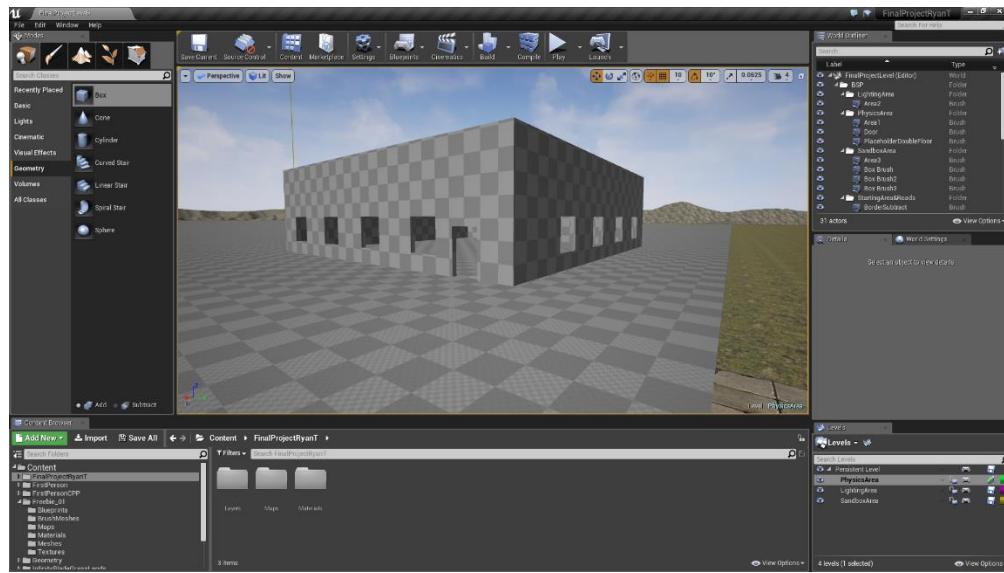


Figure 103 - Whiteboxed Building

Below, you can see a basic white texture has been added to the walls, and a basic blue texture added to the floor and ceiling.

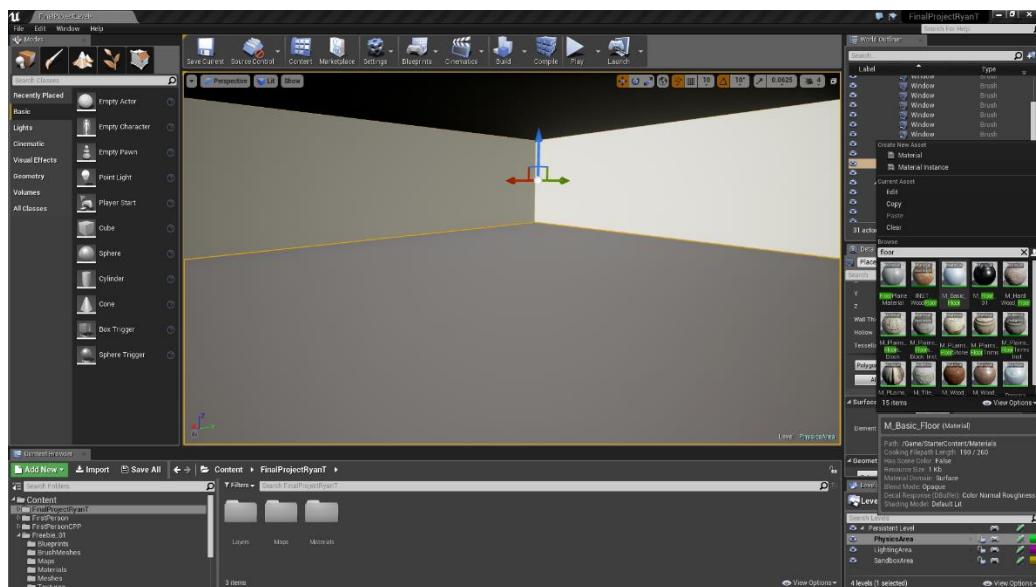


Figure 104

Below you can see a cone has been added to act as a placeholder for a ceiling light.

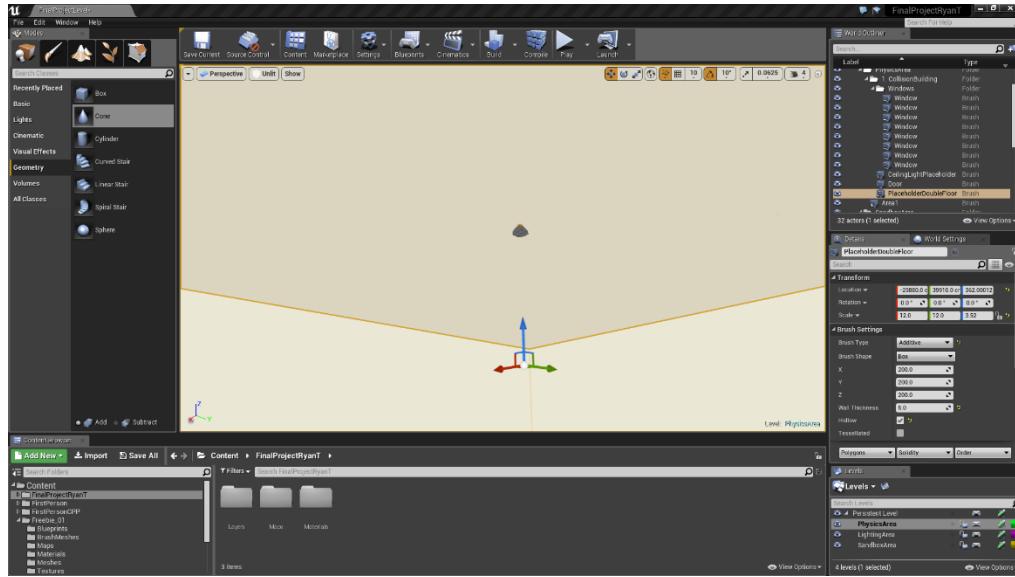


Figure 105 - Ceiling Light

The ceiling light placeholder has been duplicated four times, and four lights have been created underneath them.

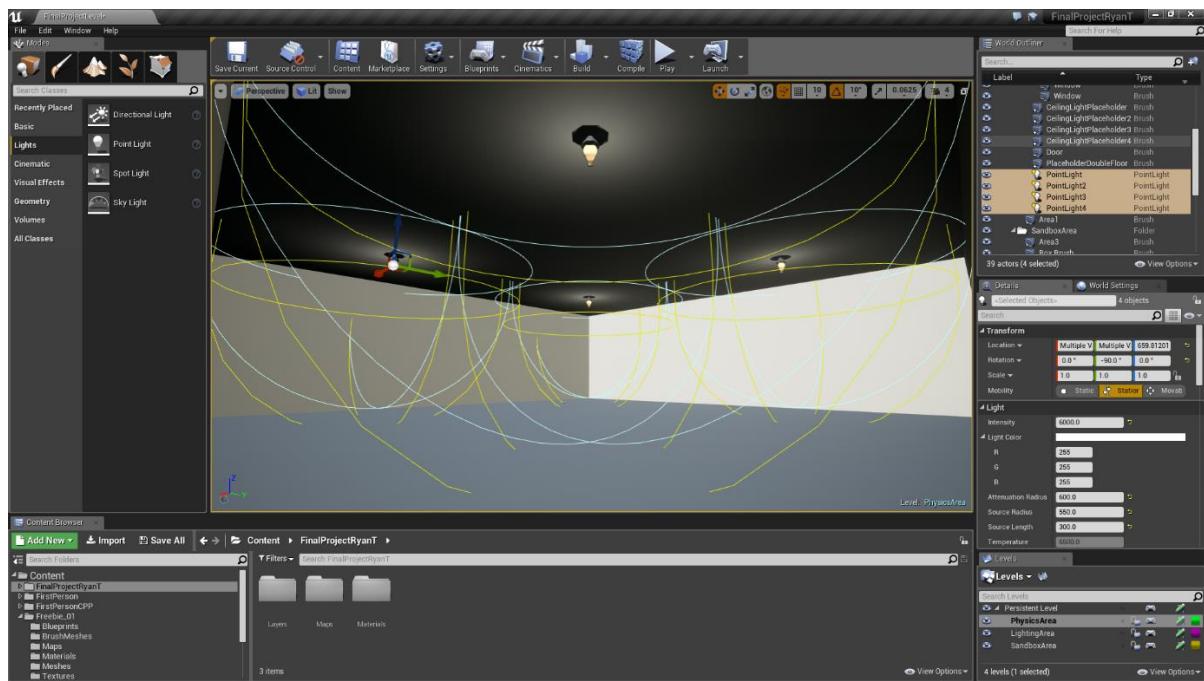


Figure 106

A post process volume has been added to the building, this is so the post processing effects can be edited, such as disabling eye adaption and bloom.

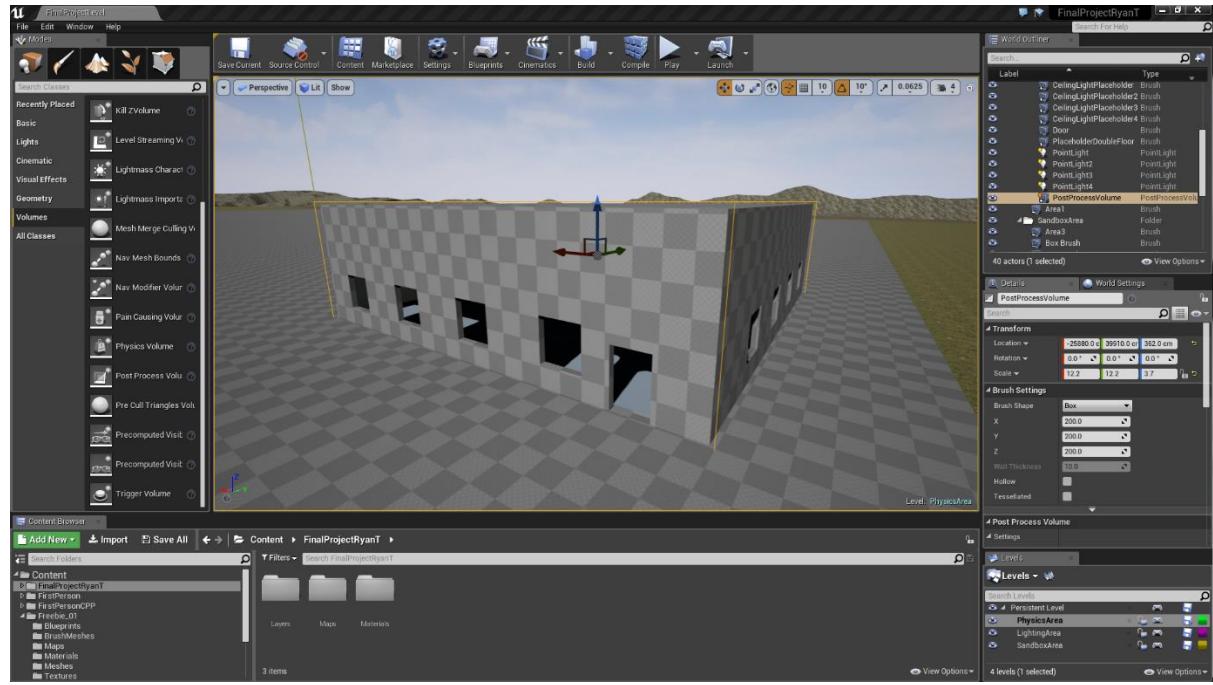


Figure 107

The point lights were ill suited for lighting the room, so they were replaced with four spotlights.

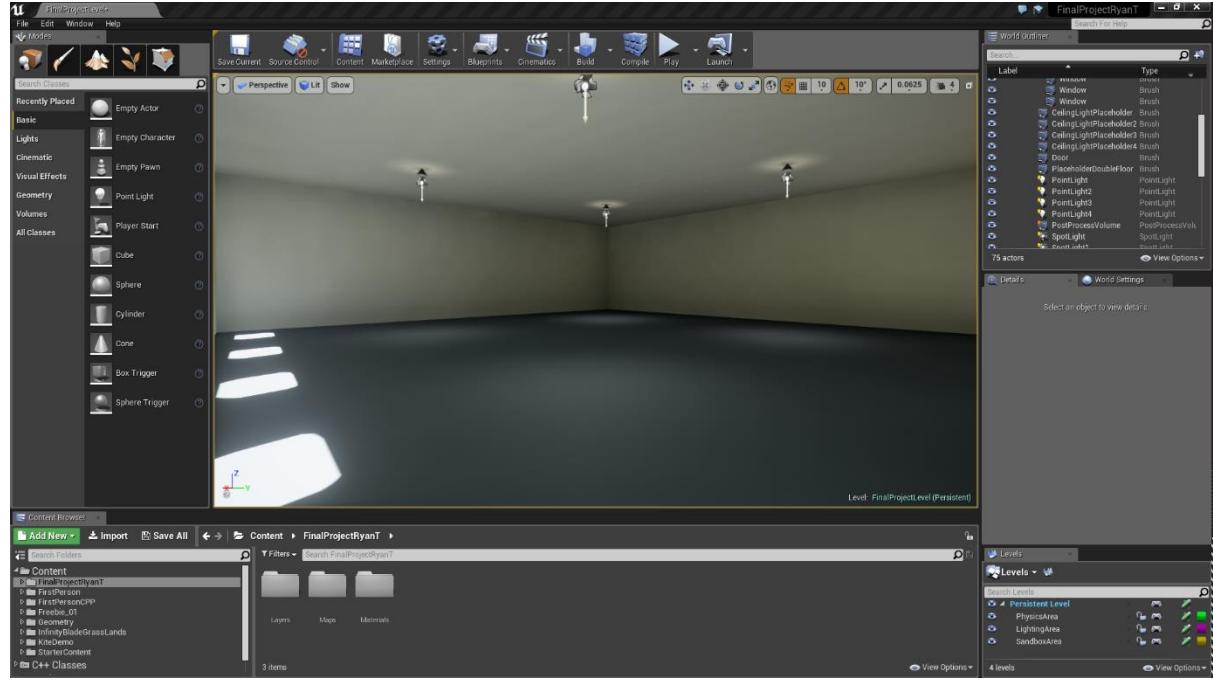


Figure 108

Below, a cube has been added into the scene, the “Simulate Physics” tickbox must be ticked in order for the shape to be able to be moved by the player.

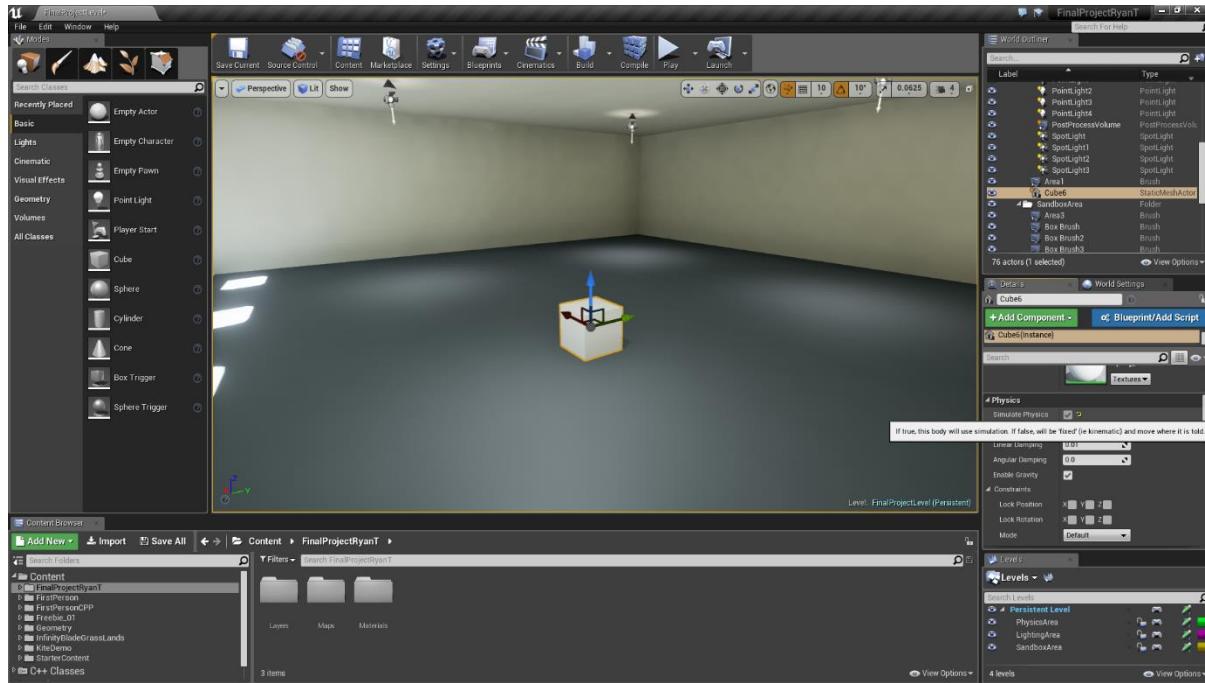


Figure 109

Here, all of the shapes have been added to the room, as per the design.

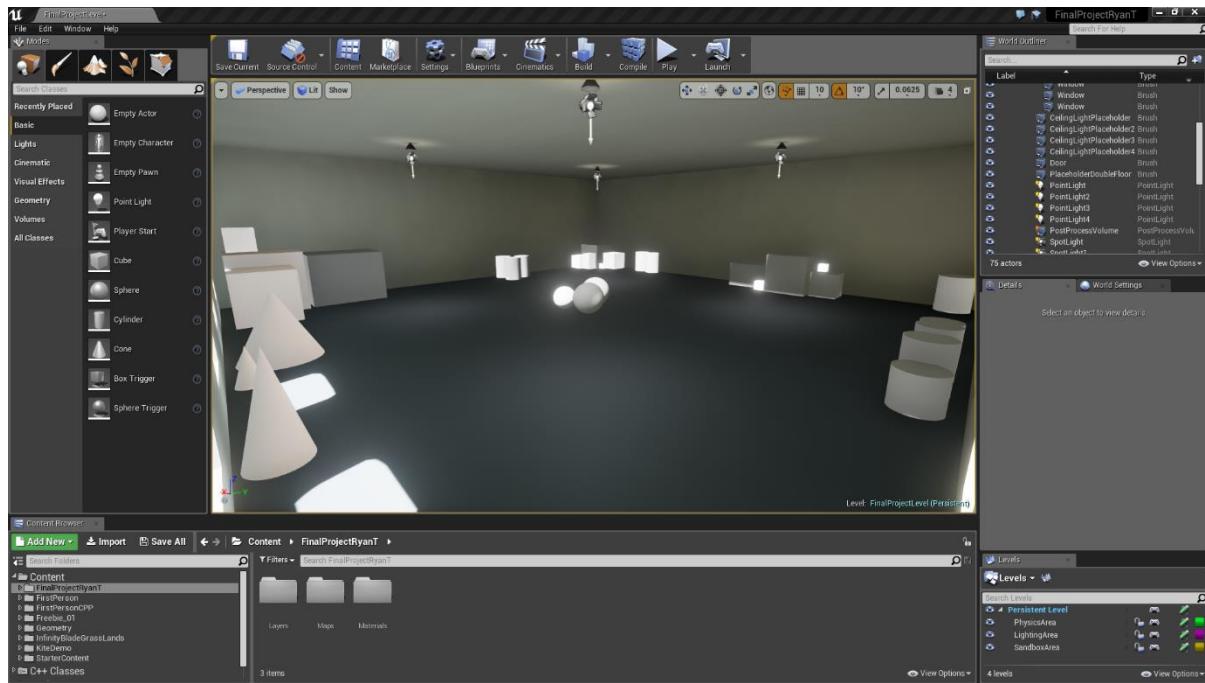


Figure 110

A barrier for the display area has also been added.



Figure 111

Below is a top down view of the building, as can be seen, the design for this room has been followed exactly.

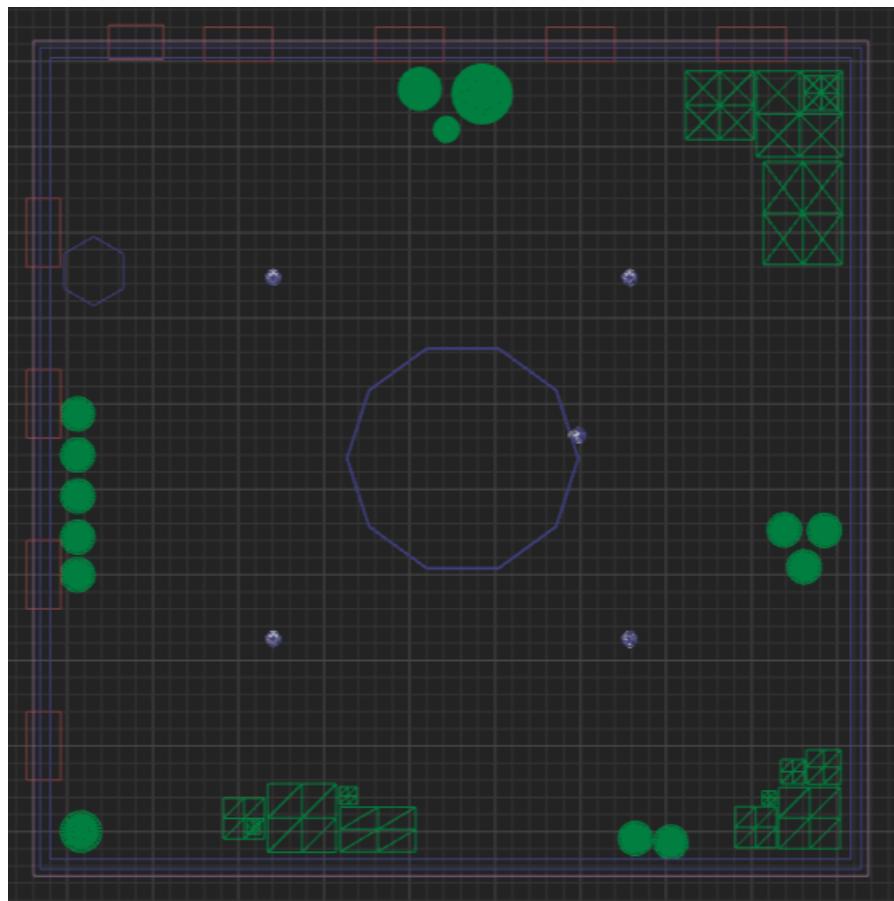


Figure 112 - Collision Building Top-Down View

In the below screenshot, you can see a new actor being created for the collision demo. This class will hold all the code and handle the functionality for the demo.

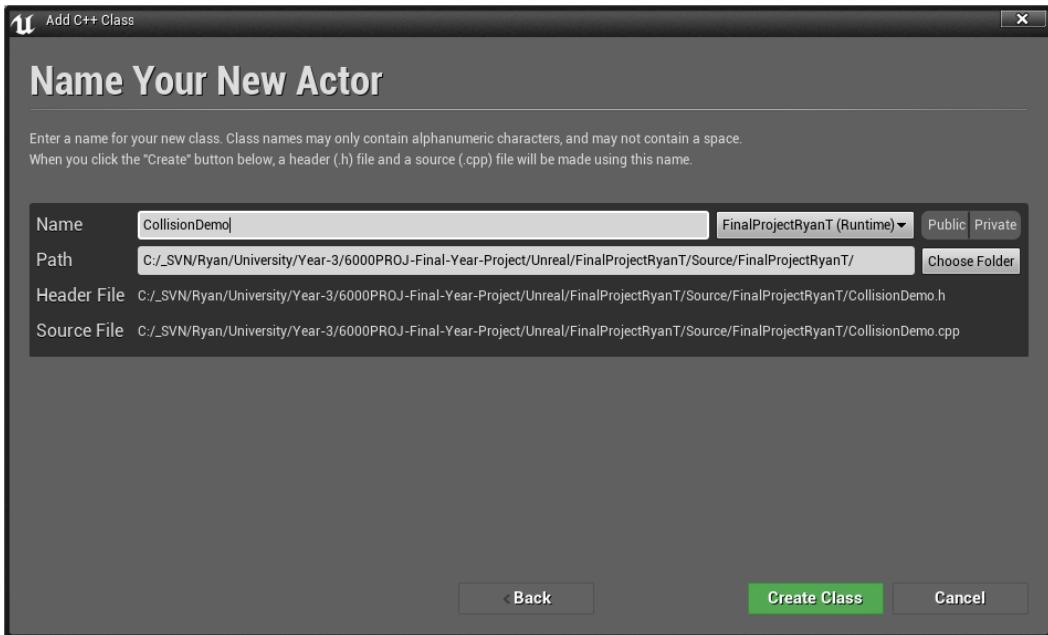


Figure 113 - New Actor Dialogue

The figure below shows the initial code for the .h file

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 [(CollisionDemo.generated.h"]
7
8 UCLASS()
9 class FINALPROJECTRYANT_API ACollisionDemo : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     ACollisionDemo();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23
24
25 };
```

Figure 114 - Initial Code .h File

The figure below shows the initial code for the .cpp file.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "CollisionDemo.h"
5
6
7 // Sets default values
8 ACollisionDemo::ACollisionDemo()
9 {
10     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
11     PrimaryActorTick.bCanEverTick = true;
12 }
13
14
15 // Called when the game starts or when spawned
16 void ACollisionDemo::BeginPlay()
17 {
18     Super::BeginPlay();
19 }
20
21
22 // Called every frame
23 void ACollisionDemo::Tick( float DeltaTime )
24 {
25     Super::Tick( DeltaTime );
26 }
27
```

Figure 115 - Initial Code .cpp File

At the bottom of the following screenshot you can see that an array of vertices has been created, these will be the tree vertices (corners) of the triangle in the demonstration/

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "CollisionDemo.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API ACollisionDemo : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     ACollisionDemo();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23 private:
24     UPROPERTY(EditAnywhere, Category = "Collision Demo", Meta = (MakeEditWidget = true))
25         TArray< FVector > LocalSpaceVerts;
26 };
```

Figure 116

Below you can see the three created vertices in the level.

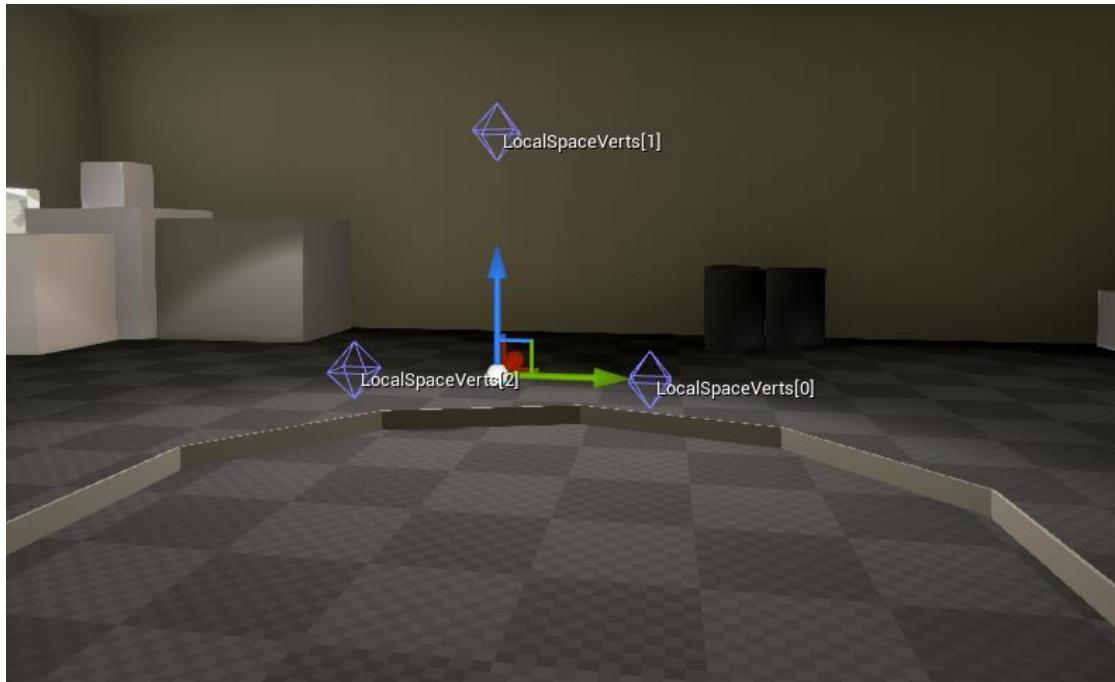


Figure 117

The below screenshot shows a new function: UpdateWorldSpaceVerts, to transform localspace positions to worldspace positions, as all calculations will be done in worldspace positions. This is called from the BeginPlay and Tick functions.

```

56
57 // ****
58 // UpdateWorldSpaceVerts - Transforms local space verts into worldspace verts
59 // ****
60 void ACollisionDemo::UpdateWorldSpaceVerts()
61 {
62     if (WorldSpaceVerts.Num() != LocalSpaceVerts.Num())
63     {
64         WorldSpaceVerts.Empty(); // Empty WorldSpaceVerts
65
66         for (int i = 0; i < LocalSpaceVerts.Num(); i++) // Added an empty vector (0,0,0) to WorldSpaceVerts, to increase it's size to match LocalSpaceVerts
67             WorldSpaceVerts.Add(FVector::ZeroVector);
68     }
69
70     FTransform Transform = GetActorTransform();
71
72     for (int i = 0; i < LocalSpaceVerts.Num(); i++)
73     {
74         FVector Pos = Transform.TransformPosition(LocalSpaceVerts[i]); // Transform each vert into World Space as verts are in actor's local space.
75
76         WorldSpaceVerts[i] = Pos;
77     }
78 }
```

Figure 118 - UpdateWorldSpaceVerts

Below you can see another new function: DrawTriangle, this function draws lines between the three vertices. It is called in the Tick function.

```

42
43 // ****
44 // DrawTriangle - Draws the triangle
45 // ****
46 void ACollisionDemo::DrawTriangle()
47 {
48     for (int i = 0; i < LocalSpaceVerts.Num(); i++)
49     {
50         int j = (i + 1) % LocalSpaceVerts.Num(); // Gets the index of the next vert, and wraps round back to the start
51                                         // if the array goes out of bounds.
52
53         DrawDebugLine(GetWorld(), WorldSpaceVerts[i], WorldSpaceVerts[j], FColor::Red, false, 0.0f, 0, 2.0f);
54     }
55 }
```

Figure 119 – DrawTriangle

Below you can see this code in action



Figure 120 - Triangle Rendered In-Game

Two lines from UpdateWorldSpaceVerts have been moved into two new functions: AddVert and ClearVerts.

```
81 // ****
82 // AddVert - Adds verts
83 // ****
84 void ACollisionDemo::AddVert()
85 {
86     WorldSpaceVerts.Add(FVector::ZeroVector);           // Added an empty vector (0,0,0) to WorldSpaceVerts,
87                                         // to increase it's size to match LocalSpaceVerts
88
89     EdgePlanes.Add(FPlane(0, 0, 0, 0));                // For each vert, add an edge plane
90 }
91
92 // ****
93 // ClearVerts - Clears all verts lists
94 // ****
95 void ACollisionDemo::ClearVerts()
96 {
97     WorldSpaceVerts.Empty();                          // Empty WorldSpaceVerts list
98
99     EdgePlanes.Empty();                            // Empty EdgePlanes list
100
101 }
```

Figure 121 - AddVert & ClearVerts

Here the two new functions are being called in UpdateWorldSpaceVerts.

```
59 // ****
60 // UpdateWorldSpaceVerts - Transforms local space verts into worldspace verts
61 // ****
62 void ACollisionDemo::UpdateWorldSpaceVerts()
63 {
64     if (WorldSpaceVerts.Num() != LocalSpaceVerts.Num())
65     {
66         ClearVerts();
67
68         for (int i = 0; i < LocalSpaceVerts.Num(); i++)
69             AddVert();
70     }
71
72     FTransform Transform = GetActorTransform();
73
74     for (int i = 0; i < LocalSpaceVerts.Num(); i++)
75     {
76         FVector Pos = Transform.TransformPosition(LocalSpaceVerts[i]);      // Transform each vert into World Space as verts are in actor's local space.
77         WorldSpaceVerts[i] = Pos;
78     }
79 }
80 }
```

Figure 122 – New Functions Being Called

Made new function to create the face plane and edge planes called UpdatePlanes.

```
105 // ****
106 // UpdatePlanes - Creates face and edge planes
107 // ****
108 void ACollisionDemo::UpdatePlanes()
109 {
110     if (WorldSpaceVerts.Num() < 3)      // If there are less than three verts, return
111         return;
112
113     // Create face plane
114     FacePlane = FPlane(WorldSpaceVerts[0], WorldSpaceVerts[1], WorldSpaceVerts[2]);
115
116     // Create edge planes
117     for (int i = 0; i < WorldSpaceVerts.Num(); i++)
118     {
119         int j = (i + 1) % WorldSpaceVerts.Num();          // Gets the index of the next vert, and wraps round back to the start
120                                         // if the array goes out of bounds.
121         FVector EdgeDir = WorldSpaceVerts[j] - WorldSpaceVerts[i];
122         EdgeDir.Normalize();
123
124         FVector EdgeNormal = FVector::CrossProduct(FacePlane, EdgeDir);
125
126         EdgePlanes[i] = FPlane(WorldSpaceVerts[i], EdgeNormal);
127     }
128 }
129
130 }
```

Figure 123 - UpdatePlanes

Here a new function has been created to draw the plane the triangle lies on, and to fill the triangle in with a colour.

```
130 // ****
131 // DrawDebugPlanes - Draws debug planes
132 // ****
133 void ACollisionDemo::DrawDebugPlanes()
134 {
135     if (WorldSpaceVerts.Num() < 3)      // If there are less than three verts, return
136         return;
137
138     FVector Pos = (WorldSpaceVerts[0] + WorldSpaceVerts[1] + WorldSpaceVerts[2]) / 3.0f;
139     FVector2D Ext(200, 200);
140     DrawDebugSolidPlane(GetWorld(), FacePlane, Pos, Ext, FColor(66, 217, 255, 10));
141
142     // Create edge planes
143     for (int i = 0; i < WorldSpaceVerts.Num(); i++)
144     {
145         int j = (i + 1) % WorldSpaceVerts.Num();          // Gets the index of the next vert, and wraps round back to the start
146                                         // if the array goes out of bounds.
147
148         Pos = (WorldSpaceVerts[i] + WorldSpaceVerts[j]) / 2.0f;
149         Ext = FVector2D(10, 100);
150
151         DrawDebugSolidPlane(GetWorld(), EdgePlanes[i], Pos, Ext, FColor::Green);
152     }
153 }
```

Figure 124 - DrawDebugPlanes

Below you can see all the previous functions and code in action.

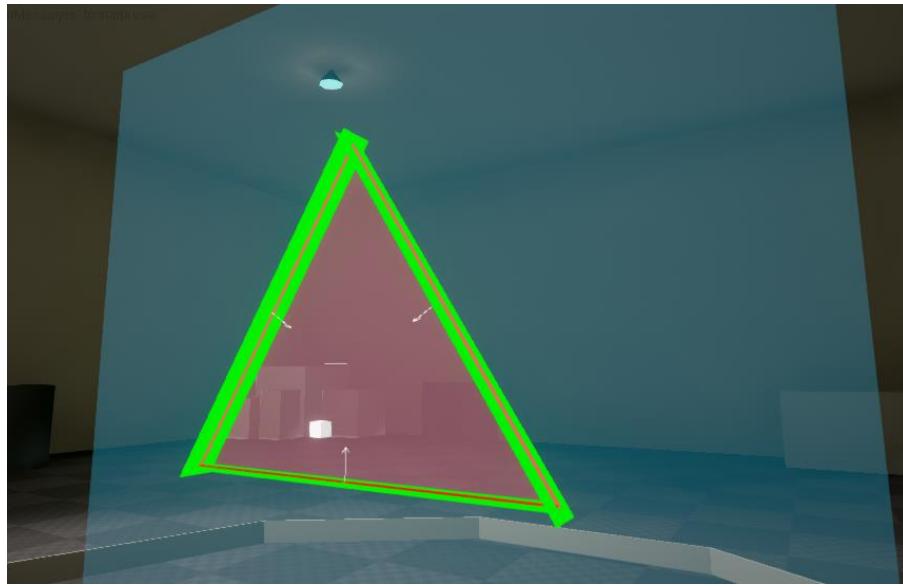


Figure 125 - Gameplay Screenshot

Below the line that will travel through the triangle, as per design, is being created. The vertices are being created in the .h file.

```
25 TArray<FVector> LocalSpaceVerts;
26
27 // UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
28 //     FVector StartPoint;
29
30 // UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
31 //     FVector EndPoint;
32
33 TArray<FVector> WorldSpaceVerts;
```

Figure 126

Here a new function has been created to perform line to triangle collision. It also displays text showing some information.

```
157 // ****
158 // PerformCollisionCheck - Checks for line to triangle collision
159 // ****
160
161 void ACollisionDemo::PerformCollisionCheck()
162 {
163     FTransform Transform = GetActorTransform();
164
165     FVector WorldSpaceStartPoint = Transform.TransformPosition(StartPoint);      // Transforms StartPoint into World Space
166     FVector WorldSpaceEndPoint = Transform.TransformPosition(EndPoint);        // Transforms EndPoint into World Space
167
168     DrawDebugLine(GetWorld(), WorldSpaceStartPoint, WorldSpaceEndPoint, FColor::White);
169
170     float DistFront = FacePlane.PlaneDot(WorldSpaceStartPoint);                // Gets the signed perpendicular distance to FacePlane (- behind plane, + ahead of plane)
171     float DistBehind = FacePlane.PlaneDot(WorldSpaceEndPoint);                 // Gets the signed perpendicular distance to EndPlane (- behind plane, + ahead of plane)
172
173     FString S = "Plane Distance (Start): ";
174     S.Append(FString::SanitizeFloat(DistFront));
175
176     S.Append("\nPlane Distance (End): ");
177     S.Append(FString::SanitizeFloat(DistBehind));
178
179     if (DistFront >= 0.0f && DistBehind <= 0.0f)                                // If the front point is infront of plane, and rear point is behind plane
180         S.Append("\n\nLINE INTERSECTING WITH PLANE!");
181
182     DrawDebugString(GetWorld(), WorldSpaceStartPoint, S, 0, FColor::White, 0.0f, true);
183
184 }
```

Figure 127 - PerformCollisionCheck

This function has been modified to calculate the intersection point of the line segment with the face plane.

```

177
178     S.Append(FString::SanitizeFloat(DistBehind));
179
180     if (DistFront >= 0.0f && DistBehind <= 0.0f)           // If the front point is infront of plane, and rear point is behind plane
181     {
182         S.Append("\n\nLINE INTERSECTING WITH PLANE!");
183
184         FVector LineVector = WorldSpaceEndPoint - WorldSpaceStartPoint;
185
186         // Calculate intersections point of line segment with face plane
187         float ProjDist = FVector::DotProduct(-LineVector, FacePlane);
188
189         float Ratio = DistFront / ProjDist;
190
191         FVector Intersection = WorldSpaceStartPoint + (LineVector * Ratio);
192
193         DrawDebugSphere(GetWorld(), Intersection, 5.0f, FColor::Green);
194         DrawDebugLine(GetWorld(), WorldSpaceStartPoint, Intersection, FColor::Red);
195     }

```

Figure 128

The function has been further modified to perform edge plain checks to determine whether the line has collided with the triangle. It also displays text showing some information.

```

196
197     DrawDebugLine(GetWorld(), WorldSpaceStartPoint, Intersection, FColor::Red);
198
199     int Count = 0;
200
201     for (int i = 0; i < EdgePlanes.Num(); i++)
202     {
203         float DistToEdge = EdgePlanes[i].PlaneDot(Intersection);           // Gets the signed perpendicular distance to current edge plane (- behind plane, + ahead of plane)
204
205         if (DistToEdge >= 0.0f)                                              // Count the number of edge plane tests that are positive
206             Count++;
207
208     }
209
210     if (Count == EdgePlanes.Num())                                         // If the point is within all edge planes (Line intersected with triangle)
211         S.Append("\n\nLINE INTERSECTING WITH TRIANGLE!");
212
213     DrawDebugString(GetWorld(), WorldSpaceStartPoint, S, 0, FColor::White, 0.0f, true);
214
215 }

```

Figure 129

Here is all the code so far in action, you can see the line intersecting with the plane, a sphere being drawn at the intersection point, and various debug information being displayed.



Figure 130 - Progress So Far

Below you can see a Text Render component being added to the demo object. This will be used to replace the debug text in the above figure.

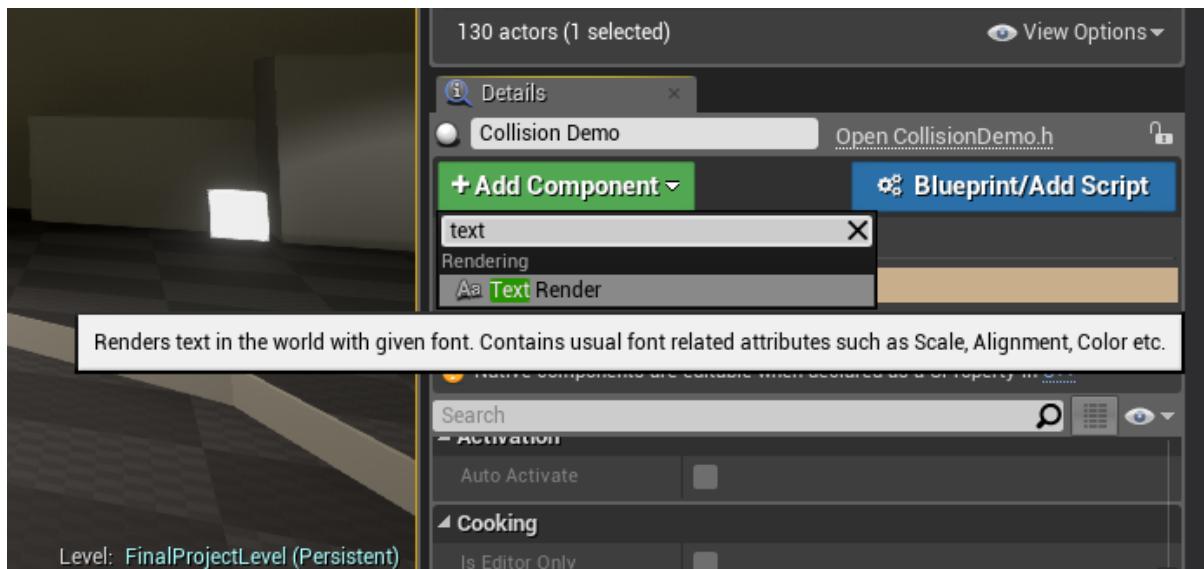


Figure 131

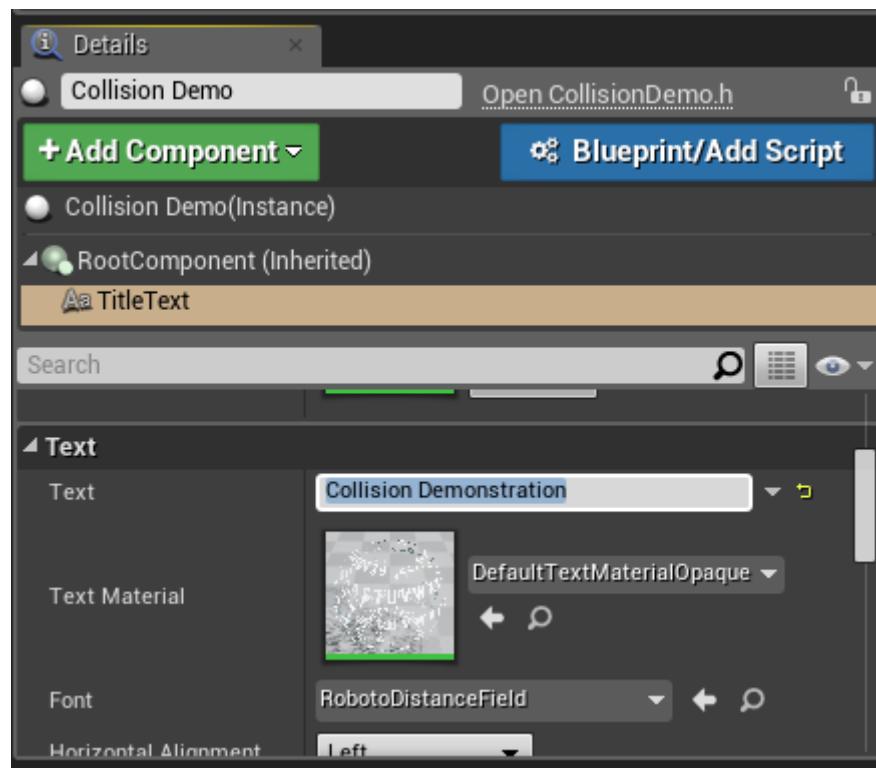


Figure 132 - Adding Text To Component



Figure 133 - Text Rendered In-Game

More text components were added to display other information in the demo.

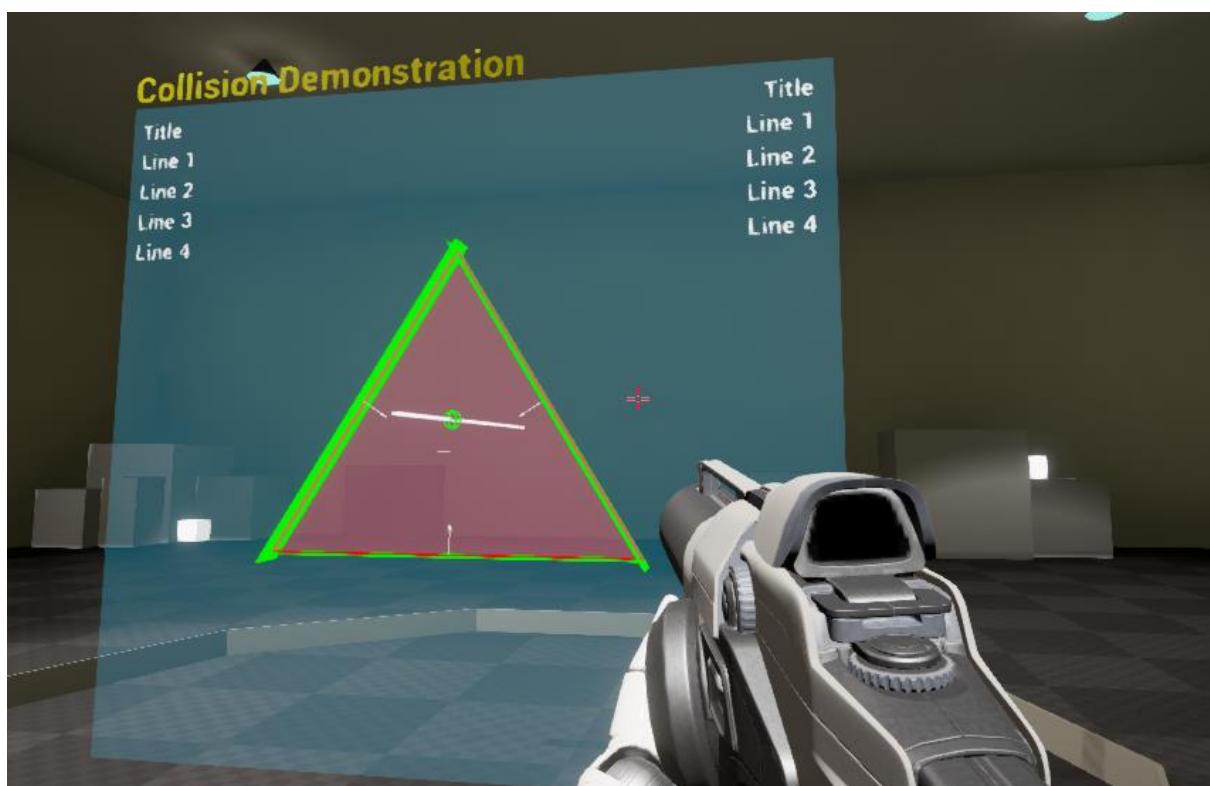


Figure 134 - More Text Rendered In-Game

The PerformCollisionCheck function has been modified to update these text components instead of the debug text.

```
197
198
199 RightTextContent = "Plane Distance (Start): ";
200 RightTextContent.Append(FString::SanitizeFloat(DistFront));
201
202 RightTextContent.Append("\nPlane Distance (End): ");
203 RightTextContent.Append(FString::SanitizeFloat(DistBehind));
204
205 if (DistFront >= 0.0f && DistBehind <= 0.0f)           // If the front point is infront of plane, and rear point is behind plane
206 {
207     LeftTextContent.Append("Line has hit plane");
208
209 FVector LineVector = WorldSpaceEndPoint - WorldSpaceStartPoint;
210
211 float ProjDist = FVector::DotProduct(-LineVector, FacePlane);          // Calculate intersections point of line segment with face plane
212
213 float Ratio = DistFront / ProjDist;
214
215 FVector Intersection = WorldSpaceStartPoint + (LineVector * Ratio);
216
217 DrawDebugSphere(GetWorld(), Intersection, 5.0f, FColor::Green);
218
219 int Count = 0;
220
221 for (int i = 0; i < EdgePlanes.Num(); i++)           // Gets the signed perpendicular distance to current edge plane (- behind plane, + ahead of plane)
222 {
223     float DistToEdge = EdgePlanes[i].PlaneDot(Intersection);
224
225     if (DistToEdge >= 0.0f)                           // Count the number of edge plane tests that are positive
226         Count++;
227
228     if (Count == EdgePlanes.Num())                   // If the point is within all edge planes (Line intersected with triangle)
229         LeftTextContent.Append("\nLine has hit triangle");
230 }
231
232 if (LeftText)
233     LeftText->SetText(FText::FromString(LeftTextContent));
234
235 if (RightText)
236     RightText->SetText(FText::FromString(RightTextContent));
237
238 }
```

Figure 135

Below you can see the text is now being updated correctly.

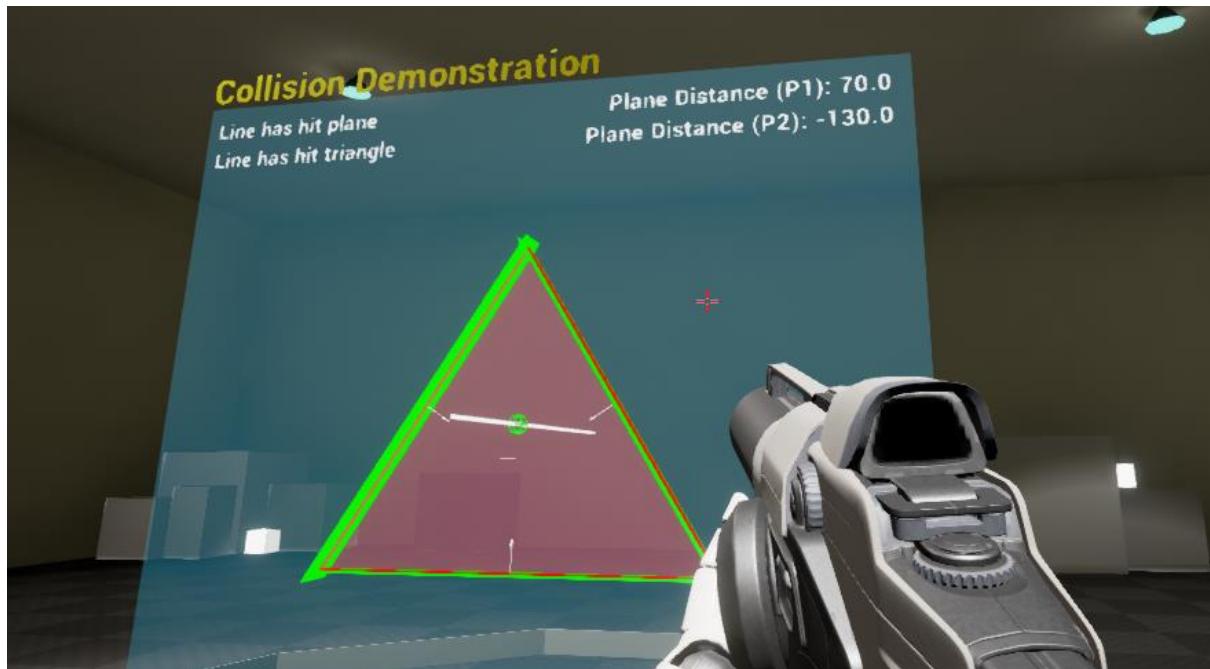


Figure 136 - Gameplay Screenshot

A USphereComponent has been added to the collision demo object. This will act as a trigger, so the demo is only activated when the player walks into it.

```
8  // ****
9  // Constructor
10 // ****
11 ACollisionDemo::ACollisionDemo()
12 {
13     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
14     PrimaryActorTick.bCanEverTick = true;
15
16     // USphereComponent is used to create a trigger volume around the actor (collision demo).
17     TriggerSphere = CreateDefaultSubobject<USphereComponent>(TEXT("SceneComponent"));
18     TriggerSphere->RegisterComponent();
19     RootComponent = TriggerSphere;
20 }
```

Figure 137

Below you can see this trigger sphere in-game.

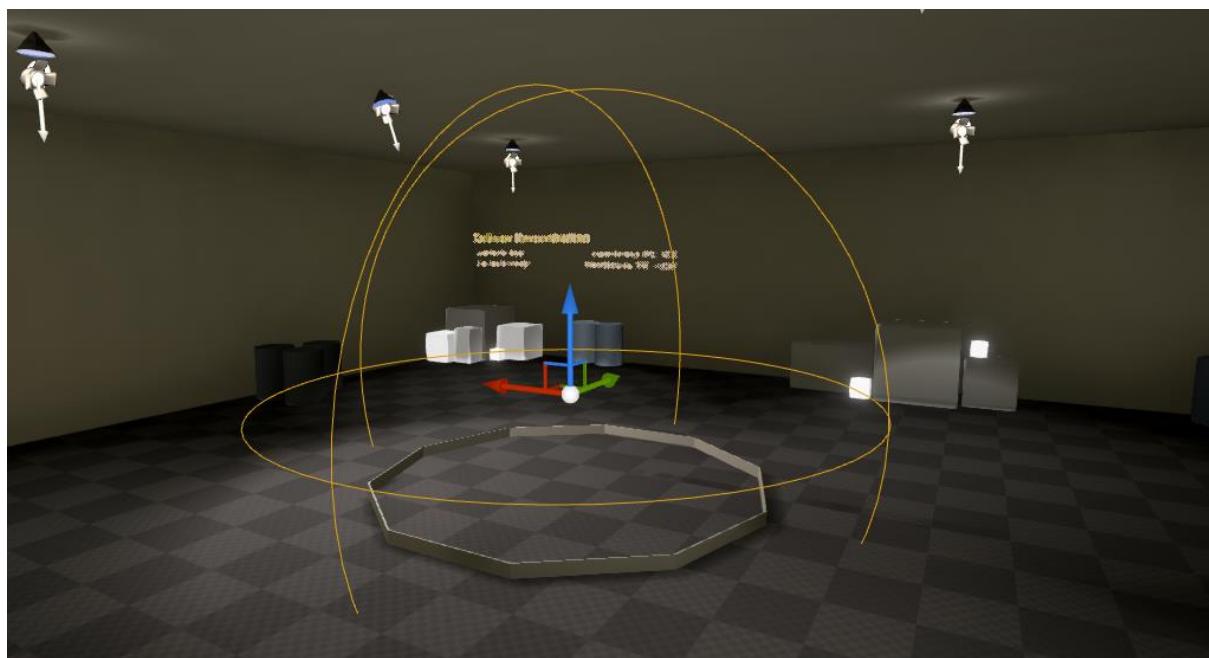


Figure 138 - Gameplay Screenshot

Below is the code that was added to the sphere to enable it to trigger the demonstration.

```
21 // ****
22 // BeginPlay - Called when the game starts or when spawned
23 // ****
24 void ACollisionDemo::BeginPlay()
25 {
26     Super::BeginPlay();
27
28     TriggerSphere->OnComponentBeginOverlap.AddDynamic(this, &ACollisionDemo::EnterSphereVolume);
29
30 }
```

Figure 139

Below is the function that is called when the player enters the sphere.

```
65 // EnterSphereVolume - Called when component enters collision volume
66 // ****
67 // ****
68 // ****
69 void ACollisionDemo::EnterSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
70 {
71     DrawDebugSphere(GetWorld(), GetActorLocation(), 20.0f, 8.0f, FColor::Orange, true, 5.0f);
72 }
73 }
```

Figure 140

Here a LevelSequence pointer has been added to the CollisionDemo.h file to allow a level sequence to be attached to the actor. A level sequence is essentially an animation in which objects in the level can be moved.

```
UPROPERTY(EditAnywhere, Category = "Collision Demo|Trigger")
ULevelSequence* LevelSequence = nullptr;
```

Figure 141

Here a SequencePlayer pointer has been added to the CollisionDemo.h to allow a level sequence to be played using engine code.

```
UPROPERTY()
ULevelSequencePlayer* SequencePlayer = nullptr;
```

Figure 142

Here the EnterSphereVolume callback function has been changed to play a level sequence (if one exists) when the trigger volume has been entered.

```
103 // ****
104 // EnterSphereVolume - Called when component enters collision volume
105 // ****
106 // ****
107 void ACollisionDemo::EnterSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
108 {
109     // Only allowed if overlapped by any kind of character
110     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
111         return;
112
113     // DrawDebugSphere(GetWorld(), GetActorLocation(), 20.0f, 8.0f, FColor::Orange, true, 5.0f);
114
115     if (LevelSequence)
116     {
117         FLevelSequencePlaybackSettings Settings;
118         Settings.LoopCount = 0;           // No looping
119         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
120         SequencePlayer->Play();
121     }
122 }
```

Figure 143

This function performs some clean-up when the level sequence is complete.

```
90 // ****
91 // OnEndSequenceEvent - Triggered when level sequence has finished playing
92 // ****
93 void ACollisionDemo::OnEndSequenceEvent()
94 {
95     SequencePlayer->Stop();
96     SequencePlayer = nullptr;
97 }
```

Figure 144

Below is the function being called; when the sequence has finished playing the clean-up function will be called.

```

50 // ****
51 // Tick - Called every frame
52 // ****
53 void ACollisionDemo::Tick( float DeltaTime )
54 {
55     Super::Tick( DeltaTime );
56
57     UpdateWorldSpaceVerts();
58
59     UpdatePlanes();
60
61     PerformCollisionCheck();
62
63     DrawTriangle();
64
65     // Sequence is not paused so Poll for sequence ending
66     if ( !SequencePlayer->IsPlaying() )
67         OnEndSequnceEvent();           // Clean up sequence player
68 }
69

```

Figure 145

Below a new level sequence is being created to animate the collision room demonstration.

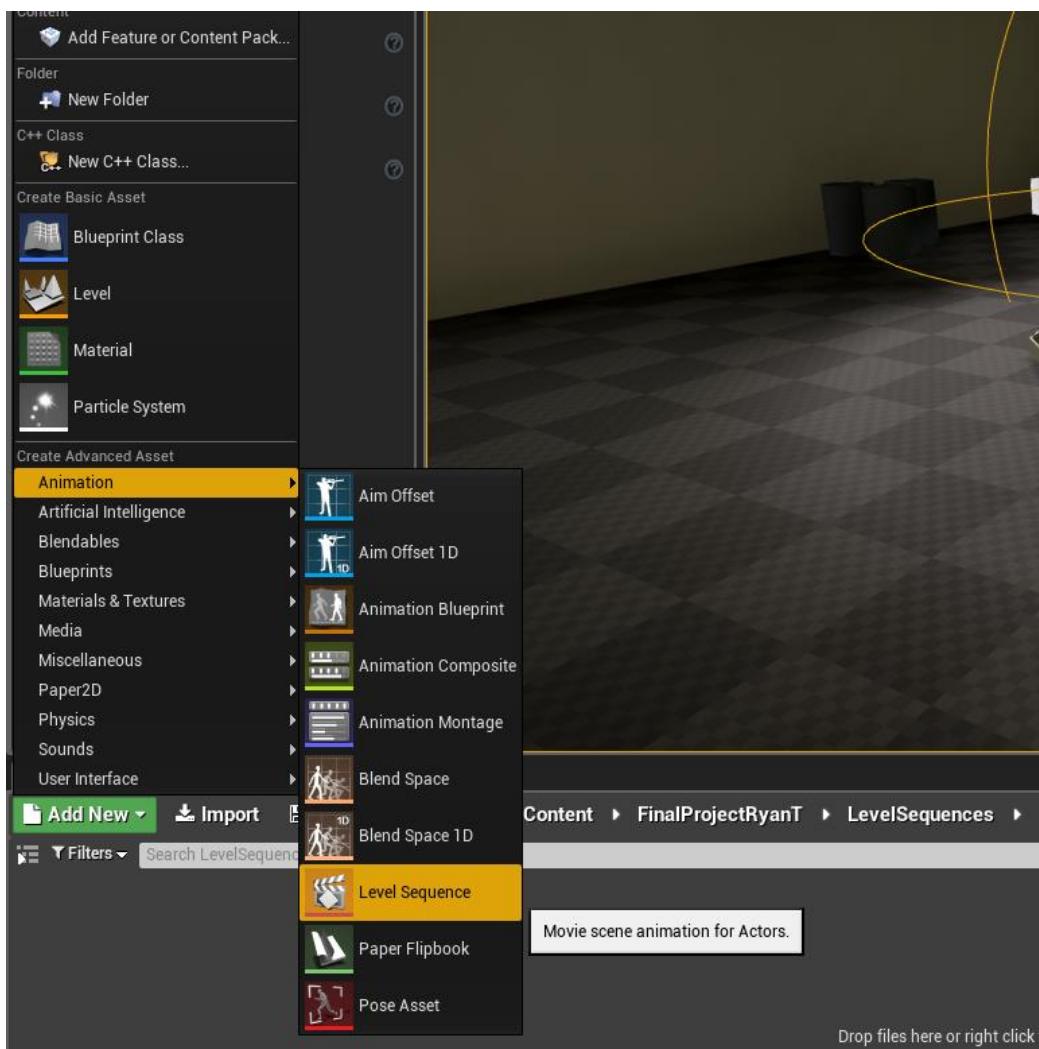


Figure 146

Here is a screenshot of the empty level sequence. The level sequencer has a timeline where you can add 'keyframes', which are basically frames in an animation that hold data about an object.

In order to animate, you select an object and place an initial keyframe, this stores that objects position. Then you move along the timeline and place another keyframe, you can then move this object and its position will be saved to that keyframe. When the animation is played, the engine will fill in everything in between and the object will animate from one keyframe to the next.

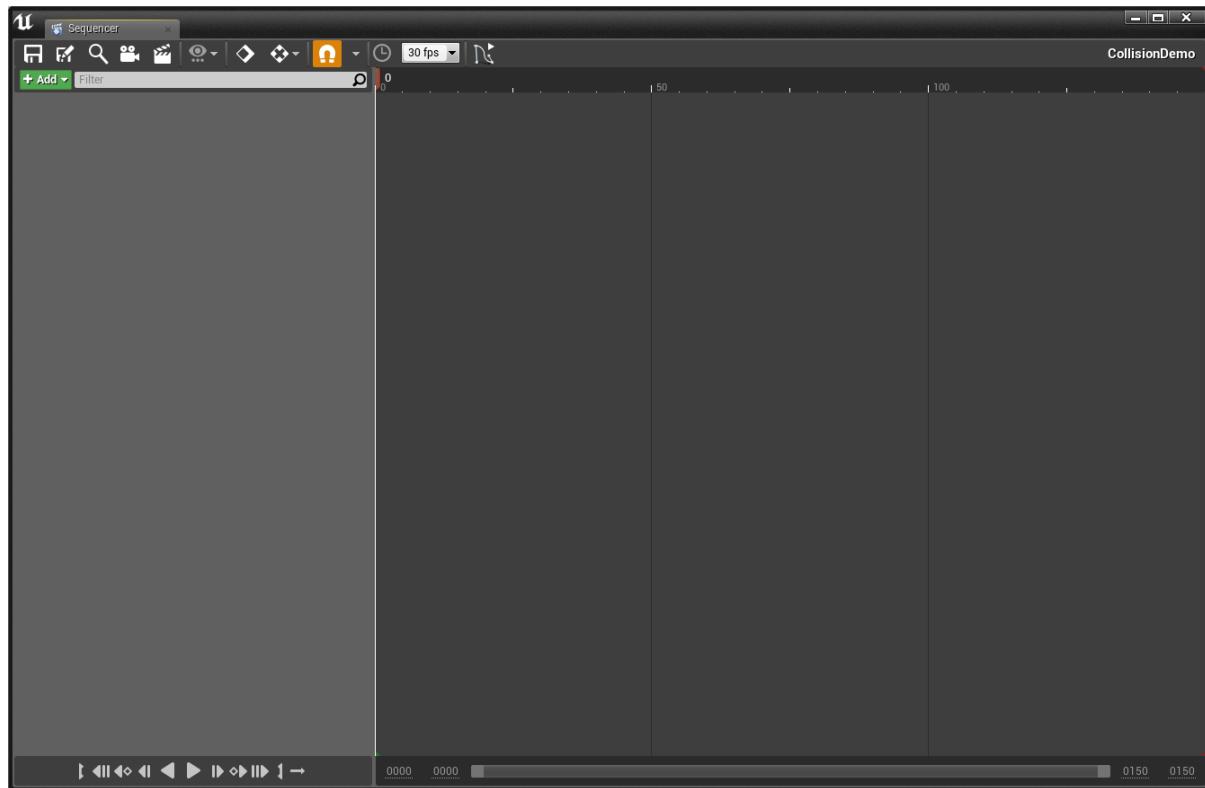


Figure 147 - Level Sequencer

Below you can see an actor being added to the sequence. This is the collision demo actor created earlier that shows the triangle and various text.

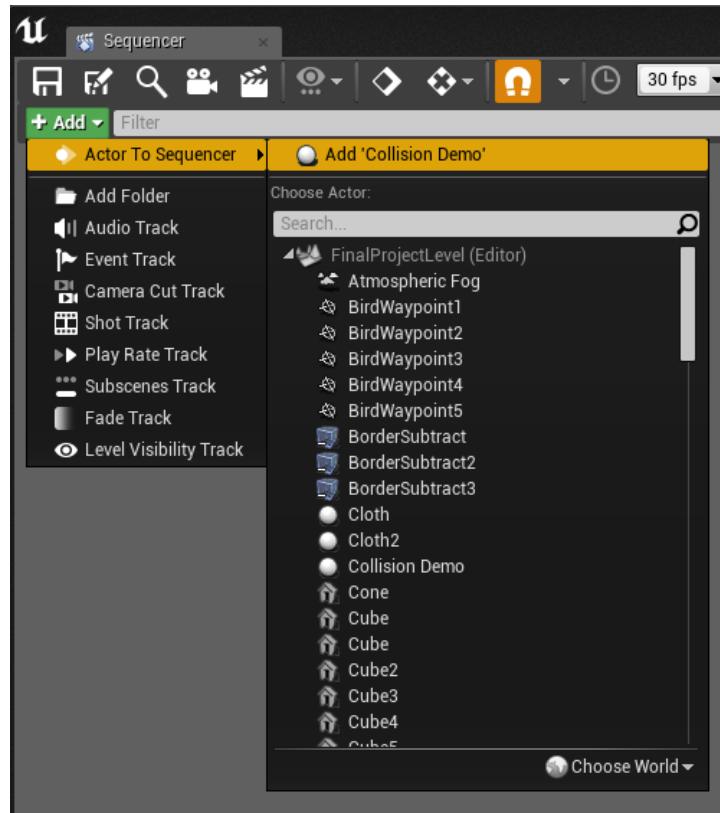


Figure 148 - Adding 'Collision Demo' To Sequence

Below is the finished level sequence for the collision demo. Further screenshots will provide further explanation into what you can see here.

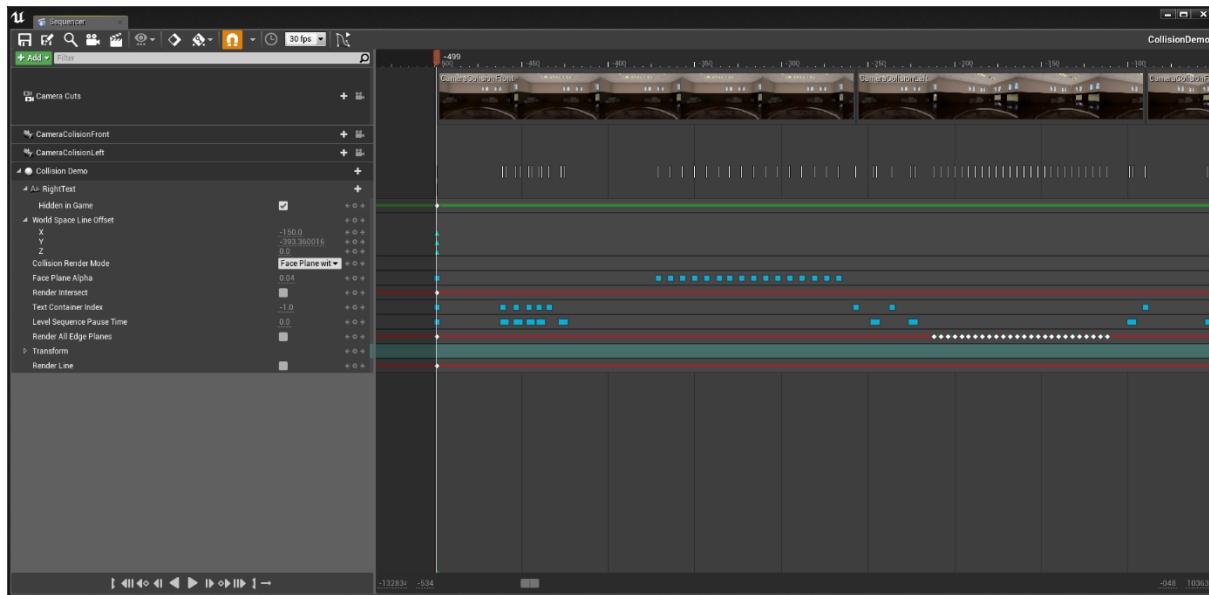


Figure 149 - Finished Collision Demo Level Sequence

The highlighted track is the 'Camera Cuts' track, this controls the cameras, in this demo it simply switches between the front and side view camera.

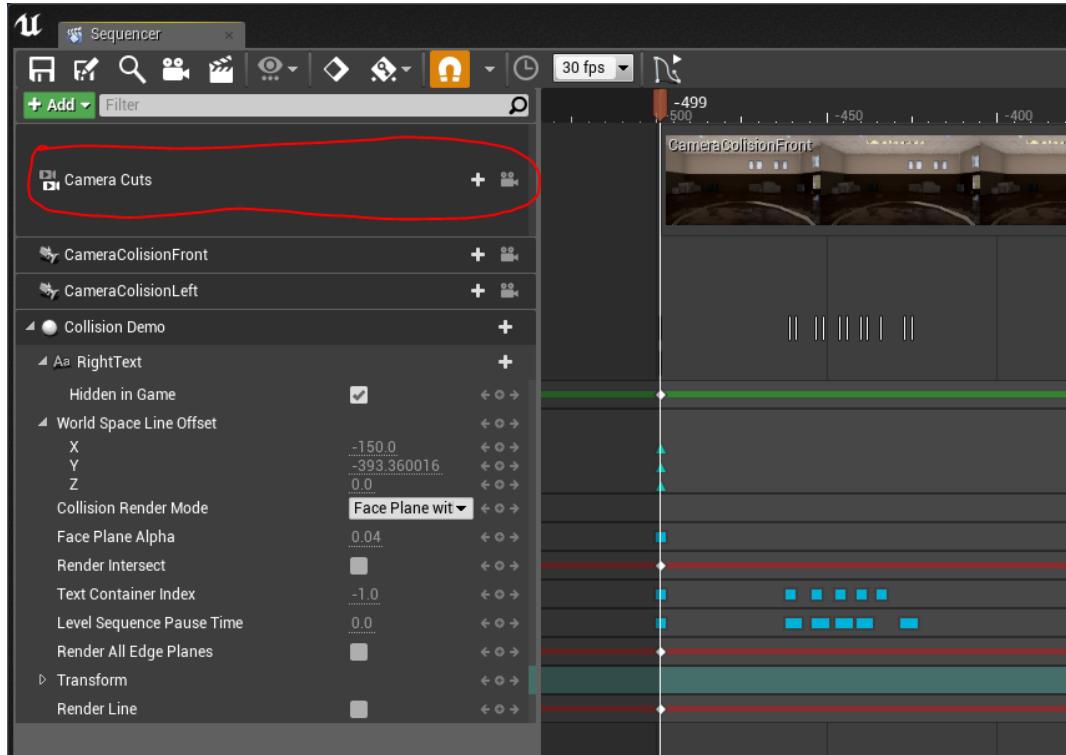


Figure 150 - Level Sequence: Camera Cuts

The highlighted area is the 'Collision Demo' track, this holds all functionality for the actual demonstration. This will be further explained below.

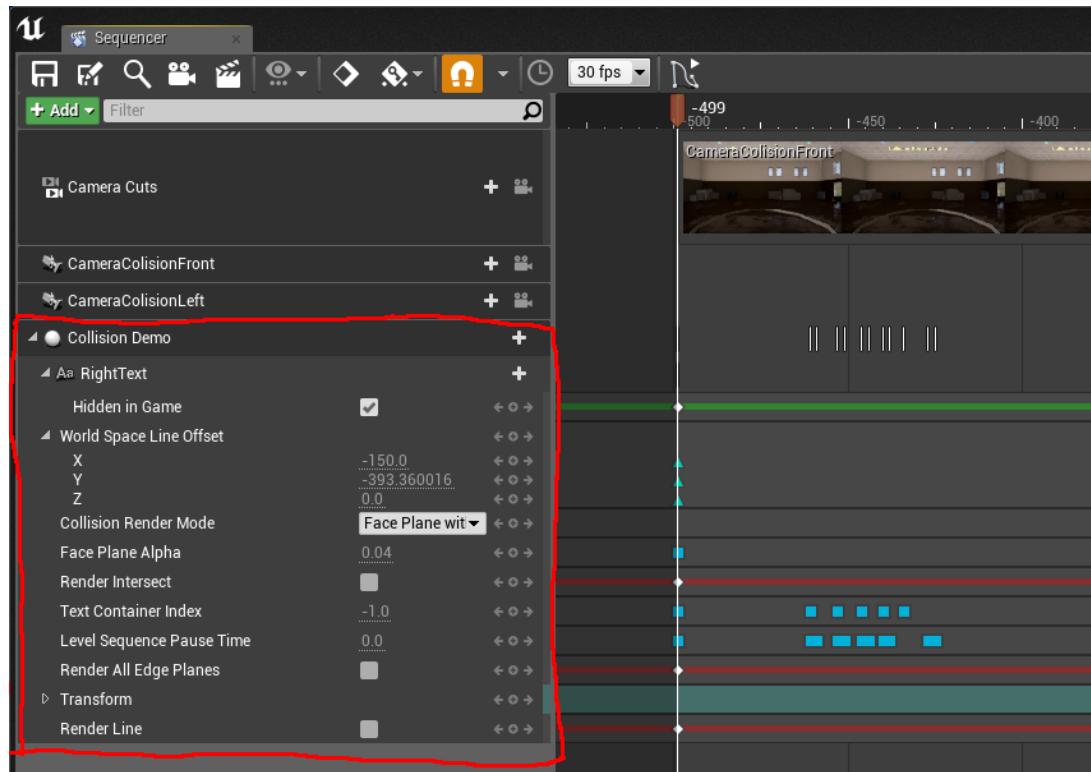


Figure 151 - Level Sequence: Collision Demo

The highlighted area is the 'RightText' track, which controls the text in the demo, specifically, when it is hidden or shown.

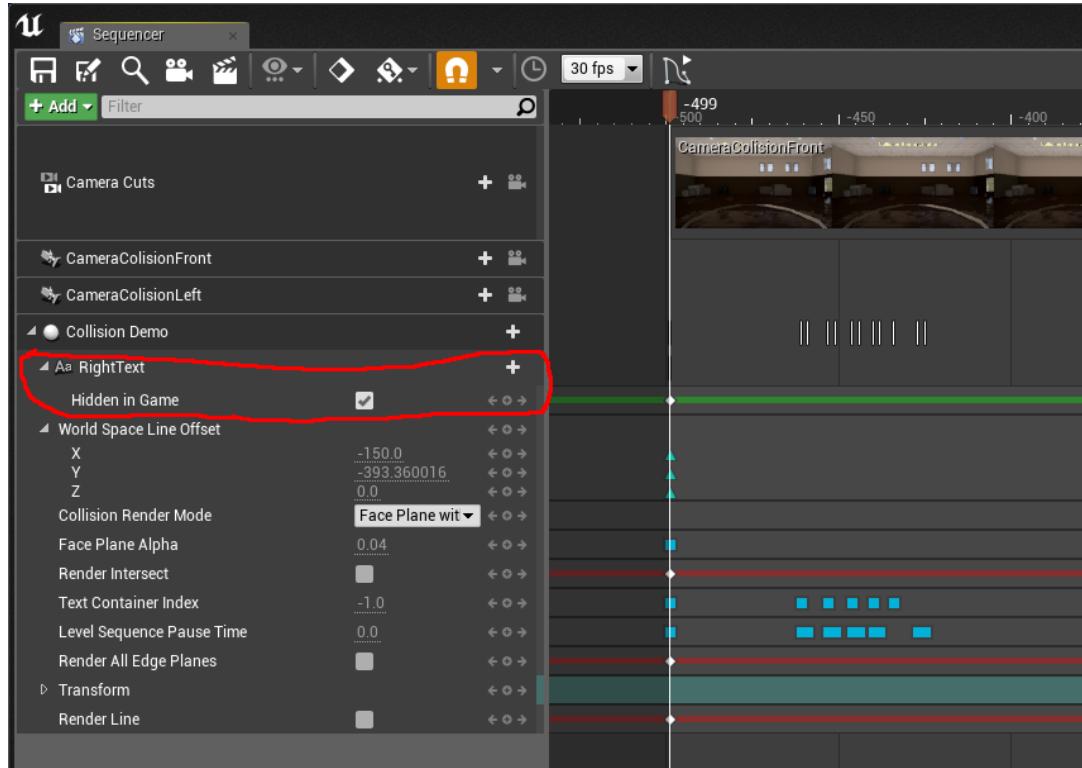


Figure 152 - Collision Demo: Right Text

The highlighted area is the 'World Space Line Offset' track, this controls the line in the demonstration; it is used to move the line around in the animation.

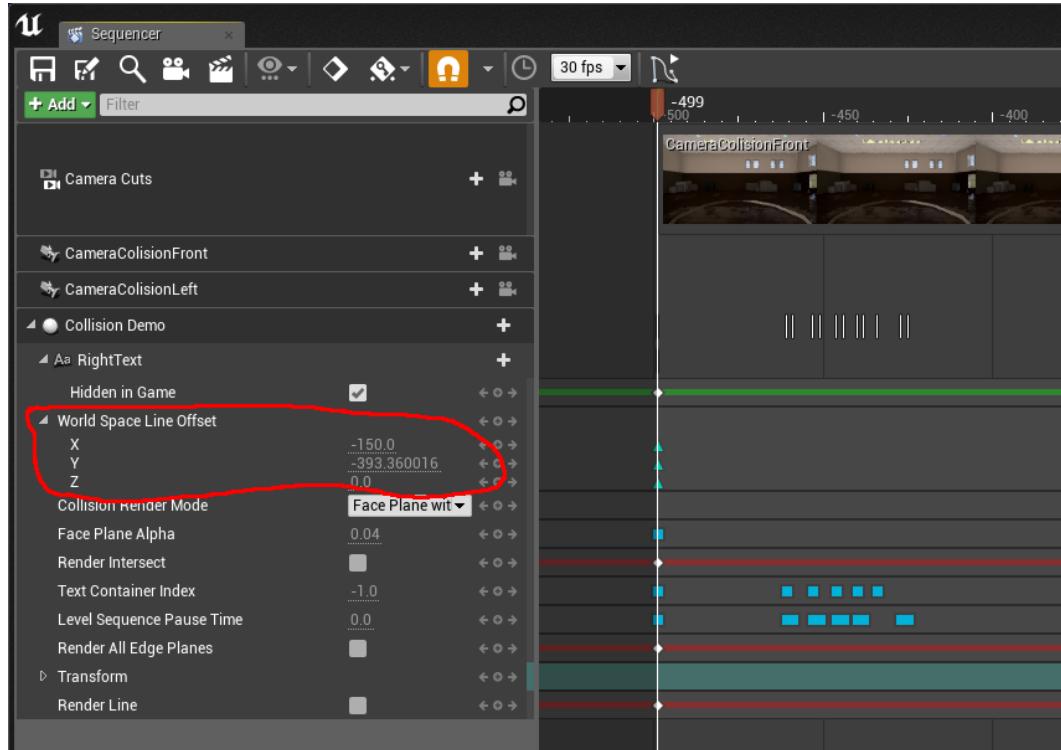


Figure 153 - Collision Demo: World Space Line Offset

The highlighted area is the 'Collision Render Mode' track. This controls when the edge planes (The green places on the three edges of the triangle) are shown or hidden.

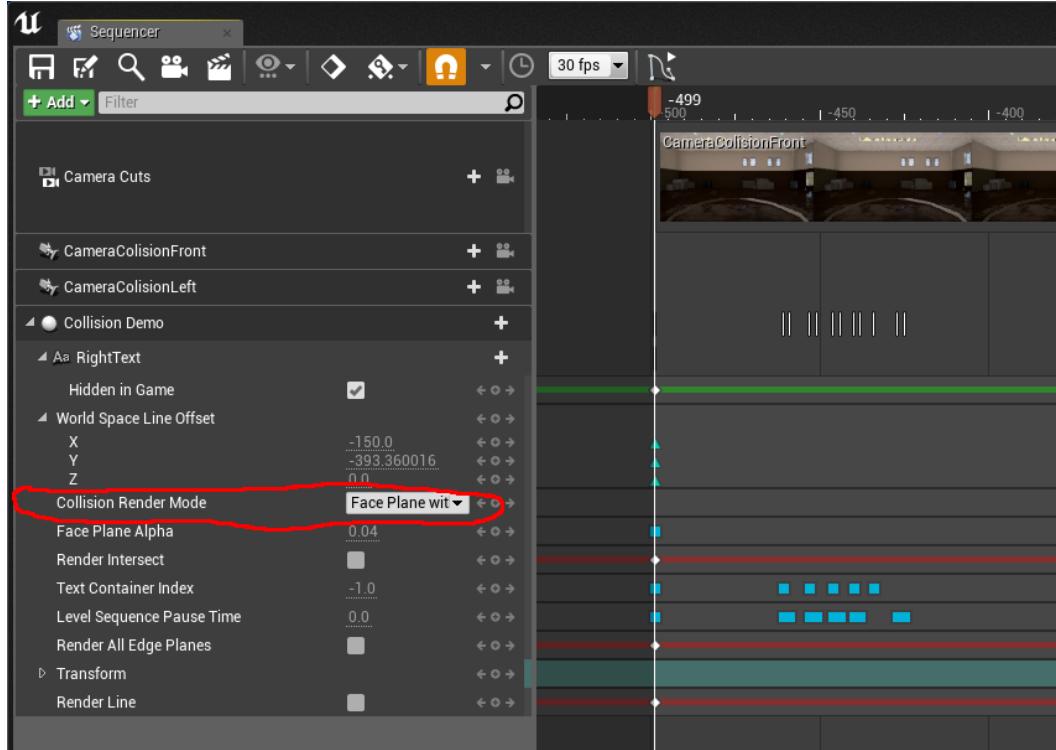


Figure 154 - Collision Demo: Collision Render Mode

The highlighted area is the 'Face Plane Alpha' track, this controls the transparency of the face plane. Allowing it to be faded in and out, or be flashed on and off.

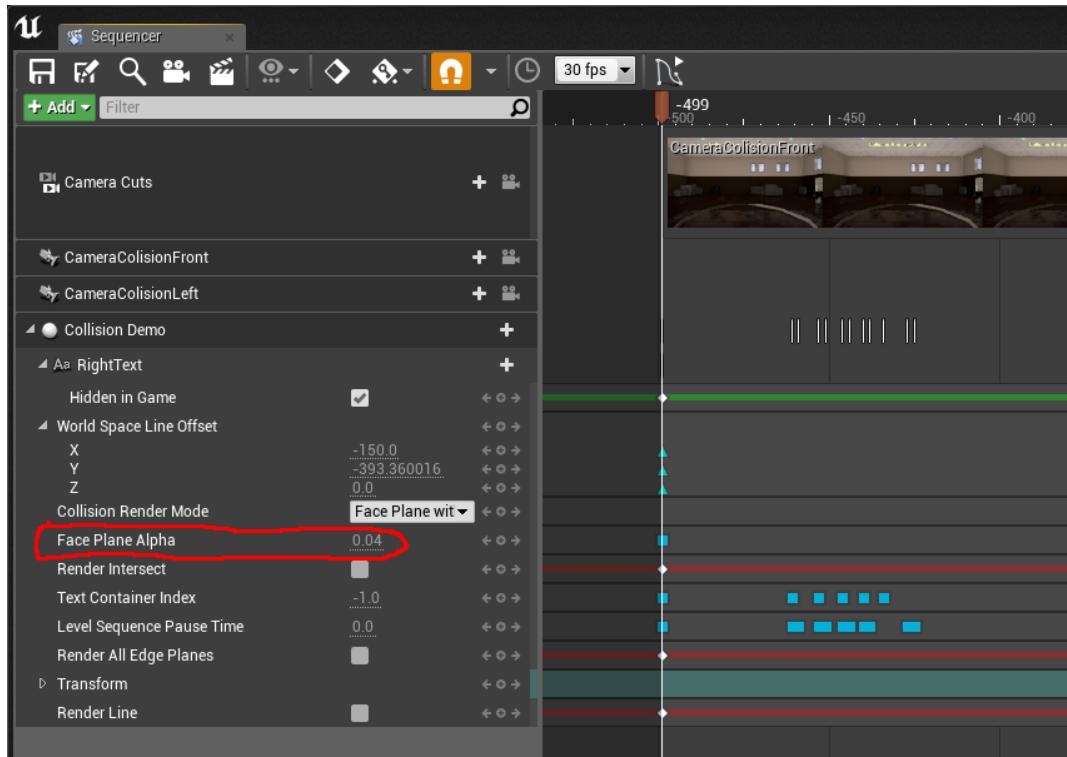


Figure 155 - Collision Demo: Face Plane Alpha

The highlighted area is the 'Render Intersect' track, this allows the sphere that is drawn on the intersect point to be turned on/off.

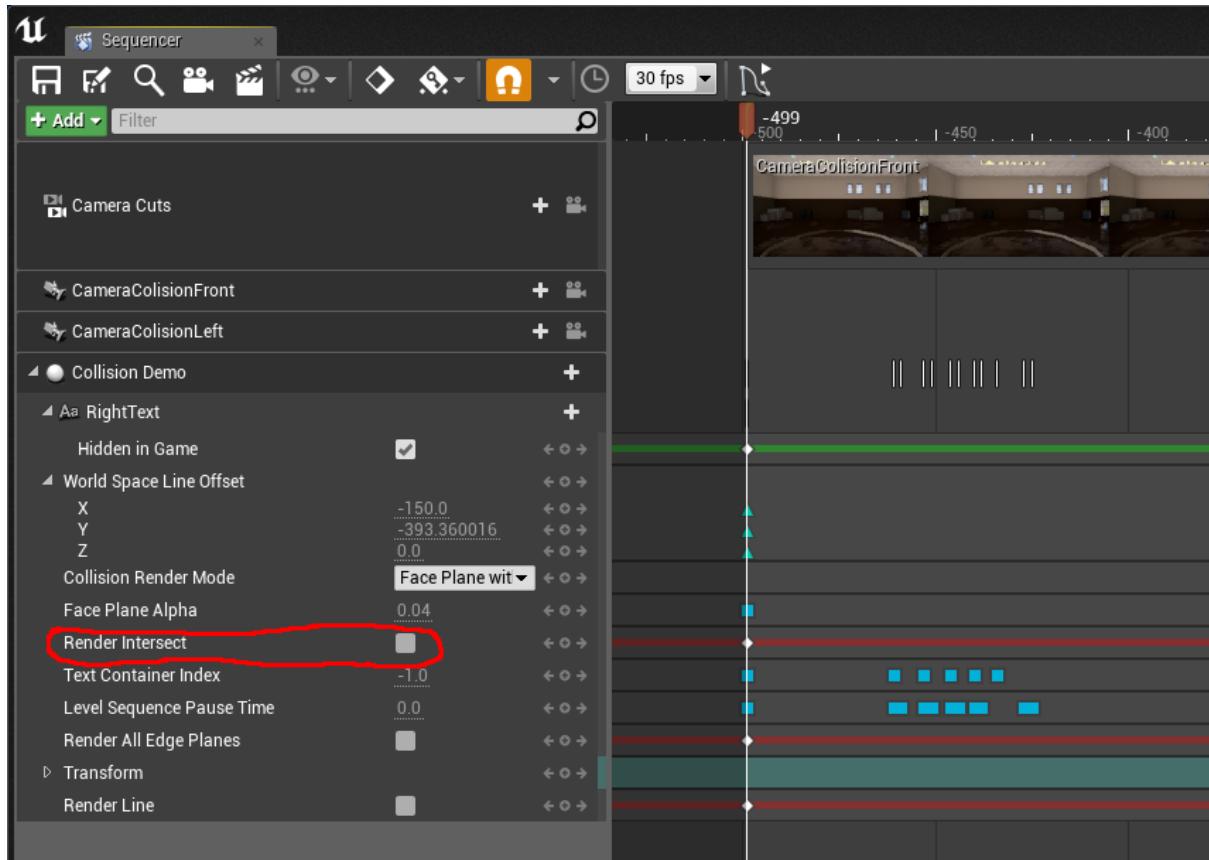


Figure 156 - Collision Demo: Render Intersect

The highlighted area is the 'Text Container Index' track, this controls the textual display for the demonstration. The text is stored in a list, this changes the index of that list, changing the text.

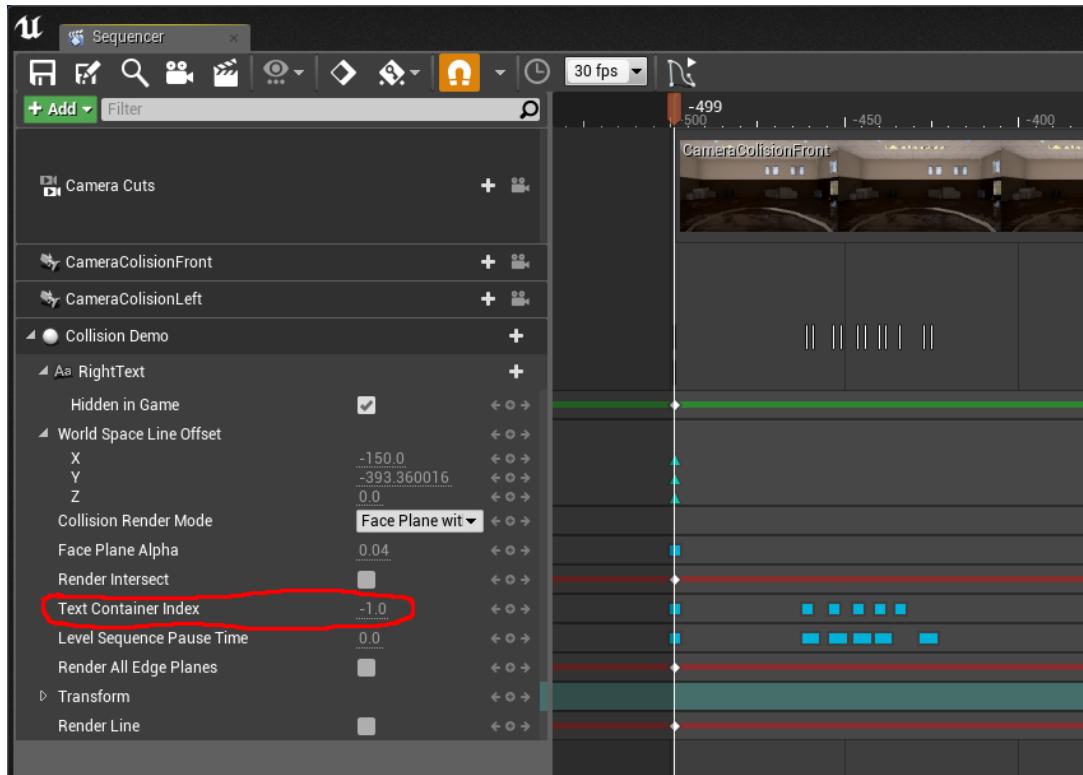


Figure 157 - Collision Demo: Text Container Index

The highlighted area is the 'Level Sequence Pause Time' track, it allows the whole level sequence to be paused, used when the text needs to describe something for example.

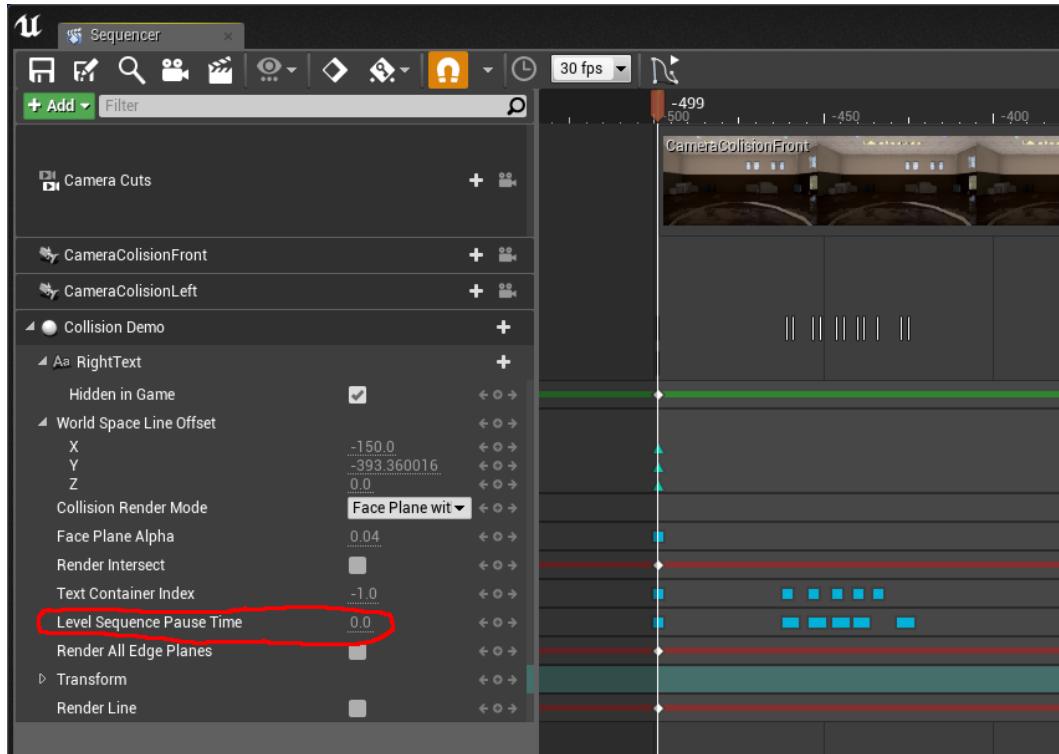


Figure 158 - Collision Demo: Level Sequence Pause Time

The highlighted area is the 'Render All Edge Planes' track, this allows the three green edge planes on the triangle to show temporarily. It is used in the start of the demo.

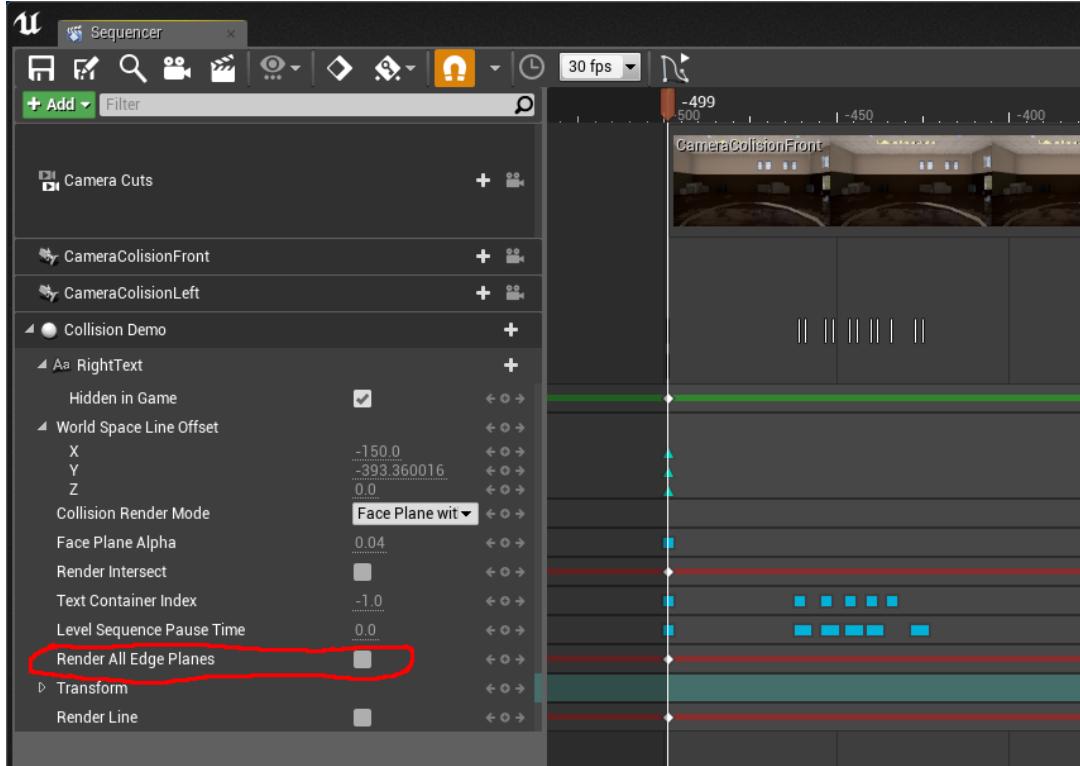


Figure 159 - Collision Demo: Render All Edge Planes

The highlighted area is the 'Render Line' track, it toggles the visibility of the line, hiding it when it is not needed.

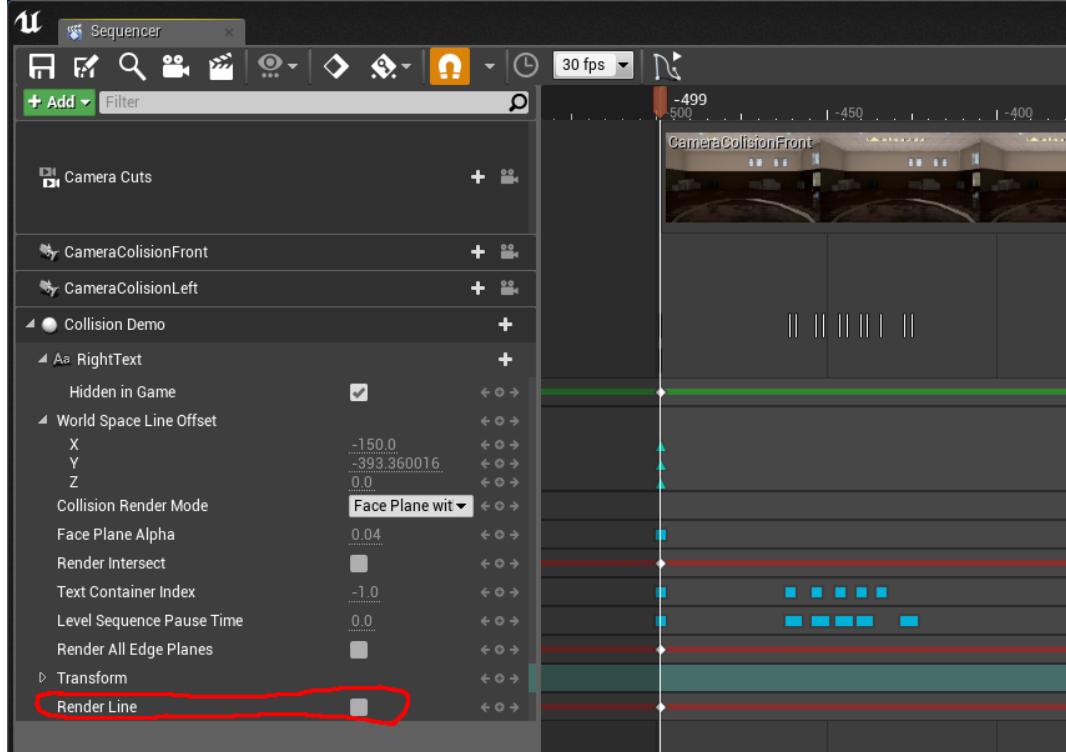


Figure 160 - Collision Demo: Render Line

The following screenshot shows the creation of the `TextContainerComponent` class. This is the class that will handle the functionality for the on-screen textual display used for the demonstrations.

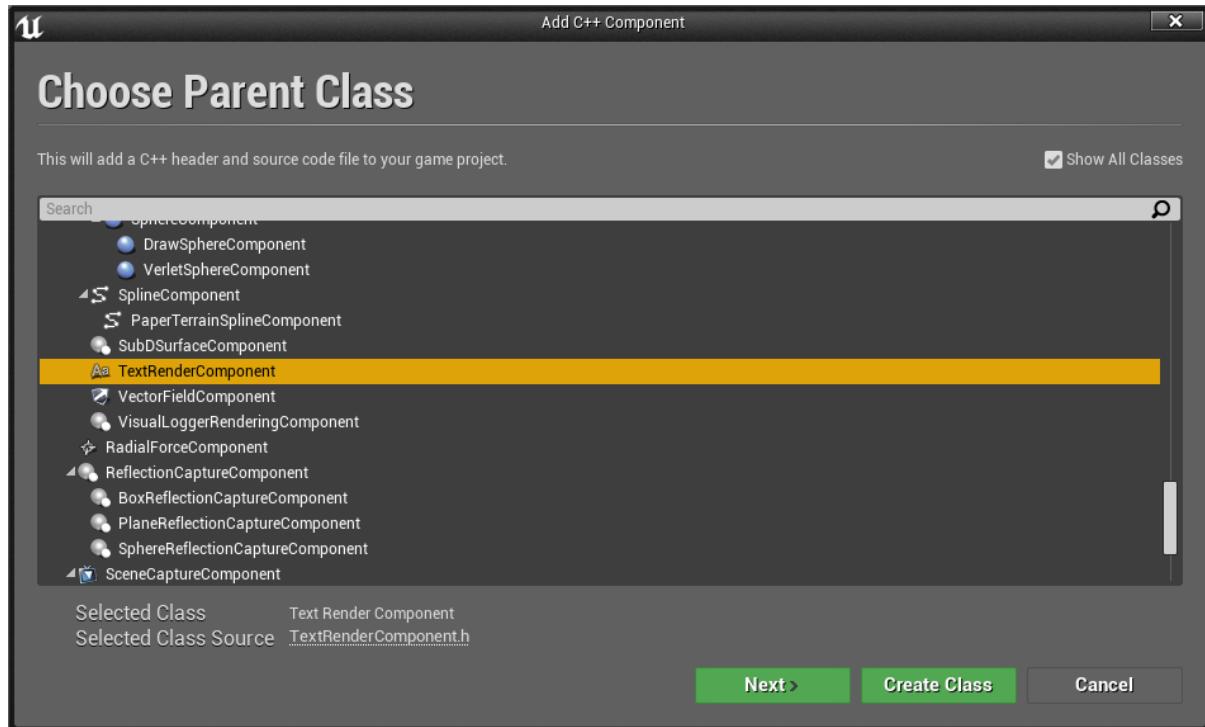


Figure 161 - Creating `TextContainerComponent` Class

Here is the dialogue for naming and choosing a path for the class.

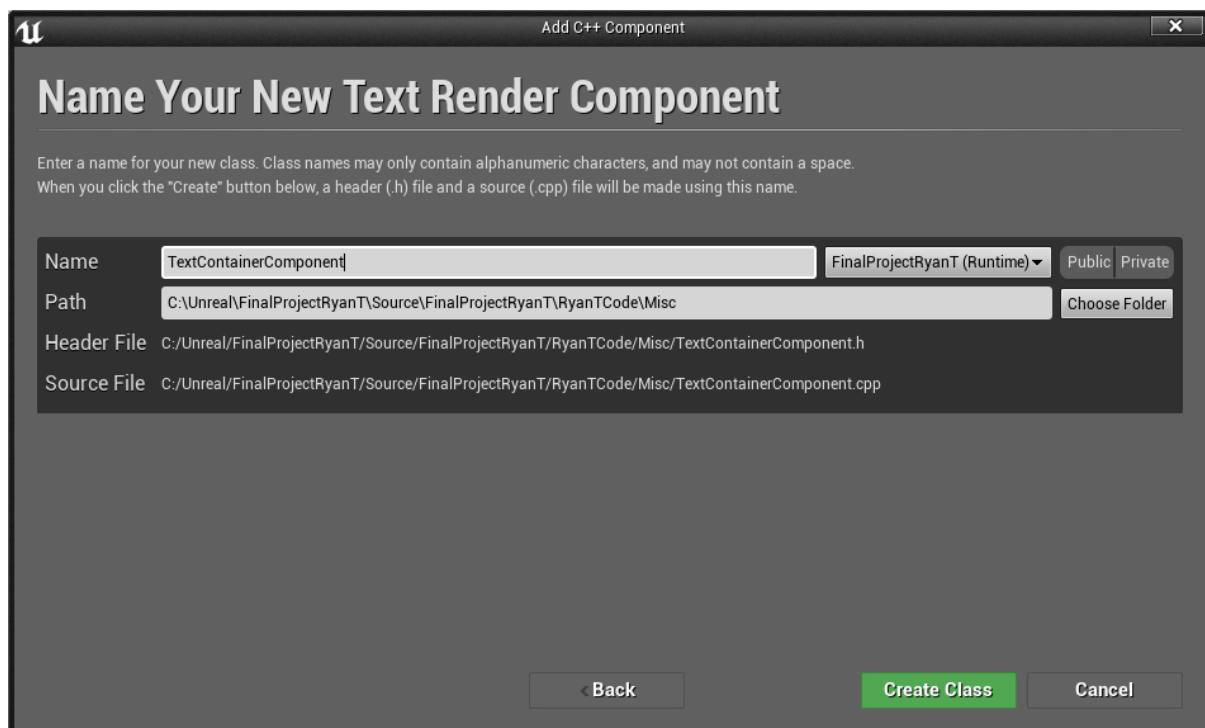


Figure 162 - Creating `TextContainerComponent` Class

The following screenshot shows the initial code for the .h file.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "Components/TextRenderComponent.h"
6 #include "TextContainerComponent.generated.h"
7
8 /**
9 *
10 */
11 UCLASS()
12 class FINALPROJECTRYANT_API UTextContainerComponent : public UTextRenderComponent
13 {
14     GENERATED_BODY()
15
16
17
18
19 };
20
```

Figure 163 - Initial Code TextContainerComponent Class .h

The following screenshot shows the initial code for the .cpp file.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "TextContainerComponent.h"
5
6
7
8
9
```

Figure 164 - Initial Code TextContainerComponent Class .cpp

The following screenshot shows the addition of a TextList array (for storing the various slides of text for the textual display), and a setter function (to set the text in the textbox) to the .h file

```
15 class FINALPROJECTRYANT_API UTextContainerComponent : public UTextRenderComponent
16 {
17     GENERATED_BODY()
18
19     bool SetTextFromList(int Index);
20
21     UPROPERTY(EditAnywhere, Category = "TextList" )
22     TArray< FString> TextList;
23 };
24
```

Figure 165

The following screenshot shows the addition of a `TextList` array (for storing the various slides of text for the textual display), and a setter function (to set the text in the textbox) to the .cpp file

```
6 // ****
7 // SetTextFromList - Set current text from list of strings that can be displayed
8 // ****
9 bool UTextContainerComponent::SetTextFromList( int Index )
10 {
11     if ( Index < 0 || Index >= Textlist.Num() )           // Bounds checking
12         return false;
13
14     SetText( FText::FromString( *Textlist[Index] ) );    // Convert FString from array to FText needed by UTextRenderComponent::SetText()
15     return true;
16 }
17
18
19
```

Figure 166

Here is a screenshot displaying the Text Container Component being added to the Collision Demo Actor.

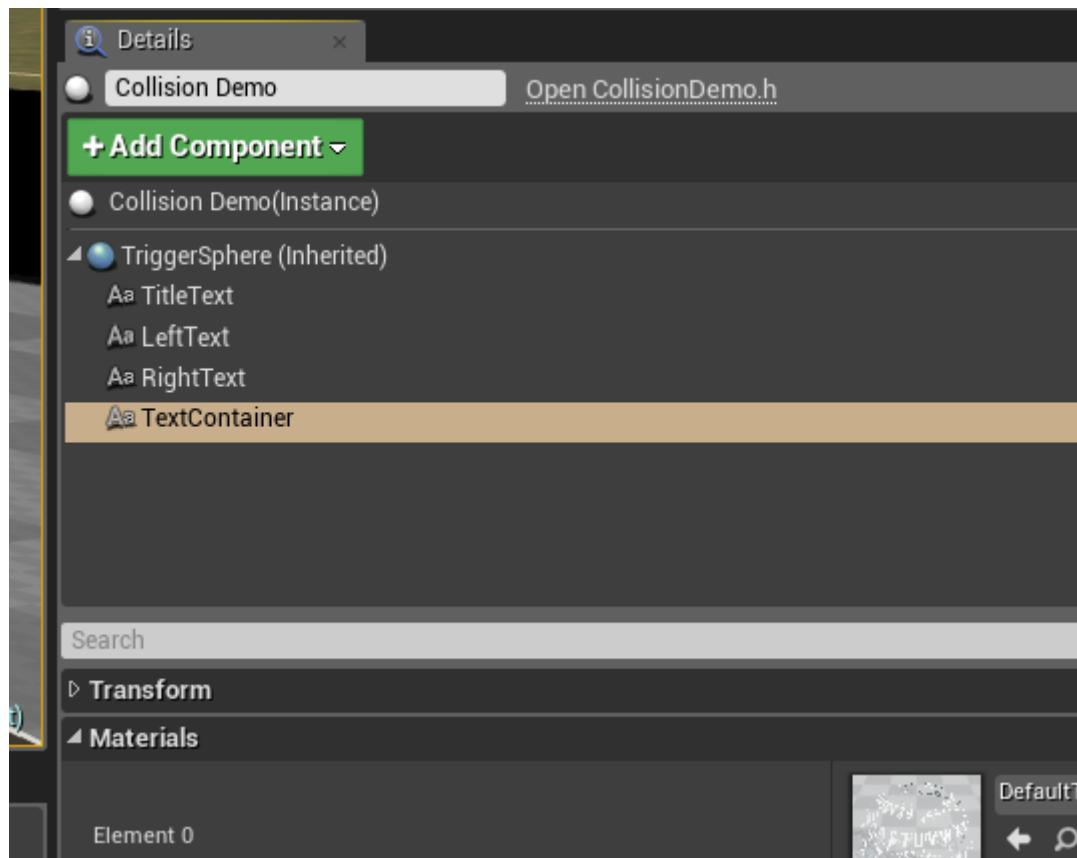


Figure 167

This screenshot displays the text that will be added into the TextContainer that will be used in this area of the demo.

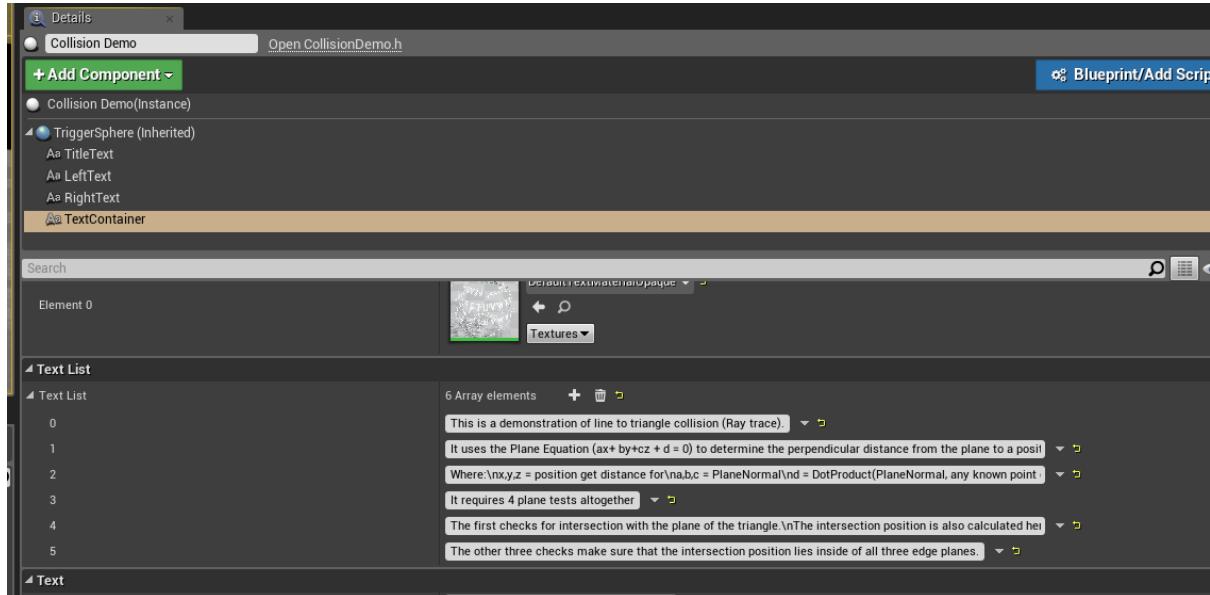


Figure 168

In this screenshot, the UTextContainer and the Text Container int had been added into the Collision Demo actor.

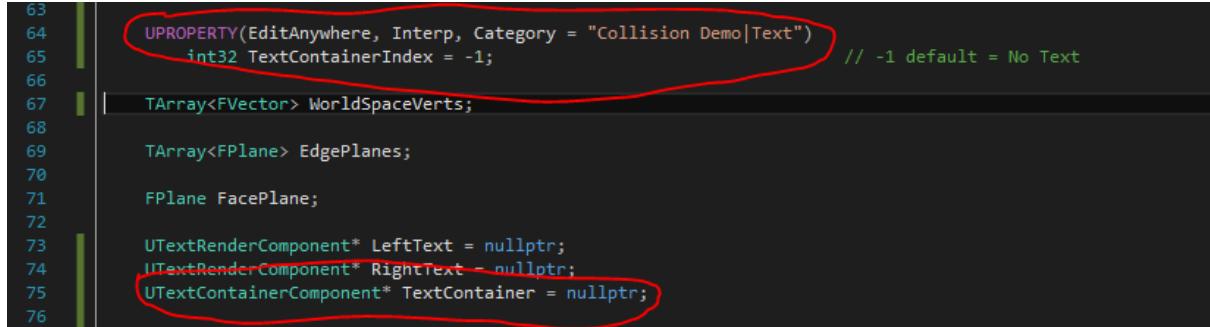


Figure 169

This screenshot a call had been added between PerformCollisionCheck function to TextContainer SetTextFromList (SetContainerIndex) in the .cpp file.

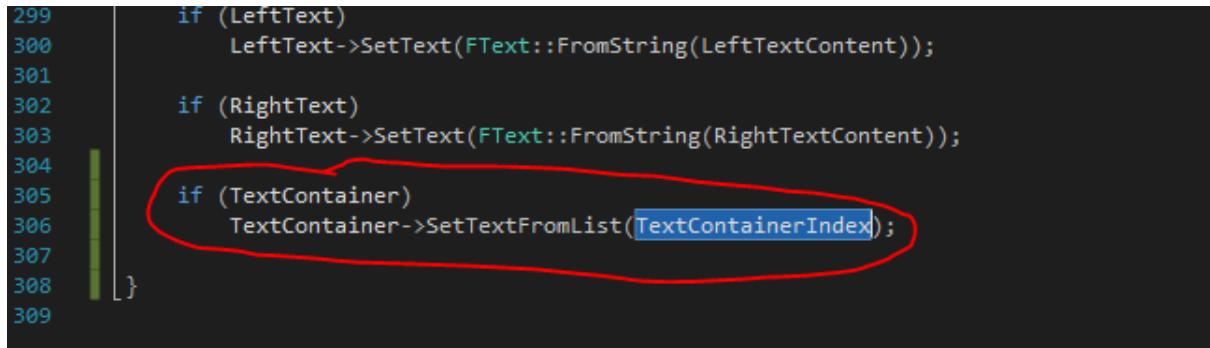


Figure 170

This screenshot displays a ‘Multiline’ meta keyword that had been added to TextList to allow Line Feeds in Details Panel.

```
23 |     private:  
24 |         UPROPERTY(EditAnywhere, Category = "TextList", meta = (MultiLine = true)) // MultiLine allows line feeds to be added in details panel usin "Shift-Enter"  
25 |             TArray< FString> TextList;  
26 |     };  
27 | 
```

Figure 171

In this screenshot, static text and a setter had been added into the .h file.

```
7 | JCLASS()  
8 | class AFinalProjectRyanTHUD : public AHUD  
9 | {  
10|     GENERATED_BODY()  
11|  
12|     public:  
13|         AFinalProjectRyanTHUD();  
14|  
15|         virtual void DrawHUD() override; // Primary draw call for the HUD  
16|     static void SetString( FString String = "" ) // Setter to set/clear sub-title text  
17|     {  
18|         HUDText = String;  
19|     }  
20|  
21|     private:  
22|         class UTexture2D* CrosshairTex; // Crosshair asset pointer  
23|  
24|         static FString HUDText; // Static text for sub-title box  
25|     };  
26|  
27|  
28|  
29| 
```

Figure 172

In this screenshot, static text and a setter had been added into the .cpp file.

```
3 | #include "FinalProjectRyanT.h"  
4 | #include "FinalProjectRyanTHUD.h"  
5 | #include "Engine/Canvas.h"  
6 | #include "TextureResource.h"  
7 | #include "CanvasItem.h"  
8 | #include "Engine.h"  
9 |  
10| FString AFinalProjectRyanTHUD::HUDText = ""; // Static text for sub-title box  
11| 
```

Figure 173

This screenshot shows the font being added into the .h file.

```
7  UClass()
8  class AFinalProjectRyanTHUD : public AHUD
9  {
10     GENERATED_BODY()
11
12    public:
13        AFinalProjectRyanTHUD();
14
15    virtual void DrawHUD() override;           // Primary draw call for the HUD
16
17    static void SetString( FString String = "" ) // Setter to set/clear sub-title text
18    {
19        HUDText = String;
20    }
21
22    private:
23        class UTexture2D* CrosshairTex;          // Crosshair asset pointer
24
25        static FString HUDText;                  // Static text for sub-title box
26
27        UFont* Font = nullptr;                  // Font used to display sub-title text
28    };
29
```

Figure 174

This screenshot shows the font being added into the .cpp file.

```
12  // ****
13  // Constructor
14  // ****
15  AFinalProjectRyanTHUD::AFinalProjectRyanTHUD()
16  {
17      // Set the crosshair texture
18      static ConstructorHelpers::FObjectFinder<UTexture2D> CrosshairTexObj(TEXT("/Game/FirstPerson/Textures/FirstPersonCrosshair"));
19      CrosshairTex = CrosshairTexObj.Object;
20
21      // Load the font used to display sub-title text
22      ConstructorHelpers::FObjectFinder<UFont> FontObject(TEXT("Font'/Game/FinalProjectRyanT/Fonts/ARIAL_18'"));
23      if (FontObject.Object)
24          Font = FontObject.Object;
25  }
```

Figure 175

This screenshot shows a function being added to draw the subtitles in the .h file.

```
6  UClass()
7  class AFinalProjectRyanTHUD : public AHUD
8  {
9  GENERATED_BODY()
10
11    public:
12        AFinalProjectRyanTHUD();
13
14    virtual void DrawHUD() override;           // Primary draw call for the HUD
15
16    static void SetString( FString String = "" ) // Setter to set/clear sub-title text
17    {
18        HUDText = String;
19    }
20
21    private:
22        void DrawSubTitleText();                // Displays description text at bottom of screen in black box
23
24    private:
25        class UTexture2D* CrosshairTex;          // Crosshair asset pointer
26
27        static FString HUDText;                  // Static text for sub-title box
28
29    };
30
```

Figure 176

This screenshot shows a function being added to draw the subtitles in the .cpp file.

```
52 // ****
53 // DrawSubTitleText - Displays description text at bottom of screen in black box
54 // ****
55 void AFinalProjectRyanTHUD::DrawSubTitleText()
56 {
57     if ( !Font || HUDText.IsEmpty() )      // If font not loaded or empty text, return
58         return;
59
60     // Width & Height of the box
61     float BGWidth = Canvas->ClipX;          // ClipX = Full screen width
62     float BGHeight = 80.0f;
63
64     // Position of the box
65     FVector2D Pos(0, Canvas->ClipY - BGHeight);
66
67     // Draw the darkened box
68     FCanvasTileItem BackgroundItem( Pos, FVector2D(BGWidth,BGHeight) ,FLinearColor(0.0f, 0.0f, 0.0f, 0.5f));
69     BackgroundItem.BlendMode = SE_BLEND_Translucent;
70     Canvas->DrawItem( BackgroundItem );
71
72     // Draw the text
73     FCanvasTextItem TextItem( Pos + FVector2D(4.0f,4.0f), FText::FromString(HUDText), Font, FLinearColor(0.8f, 0.8f, 0.8f, 1.0f) );
74
75     // Enable drop shadow
76     // TextItem.FontRenderInfo.bEnableShadow = true;
77     // TextItem.ShadowColor = FLinearColor(0.0f,0.0f,0.0f,1.0f);
78
79     TextItem.ShadowOffset = FVector2D( 1.0f, 1.0f );
80     TextItem.BlendMode = SE_BLEND_Translucent;
81     TextItem.Scale = FVector2D( 0.8f, 0.8f );
82
83     Canvas->DrawItem( TextItem );
84 }
```

Figure 177

The screenshot below shows a sign being added into the level.

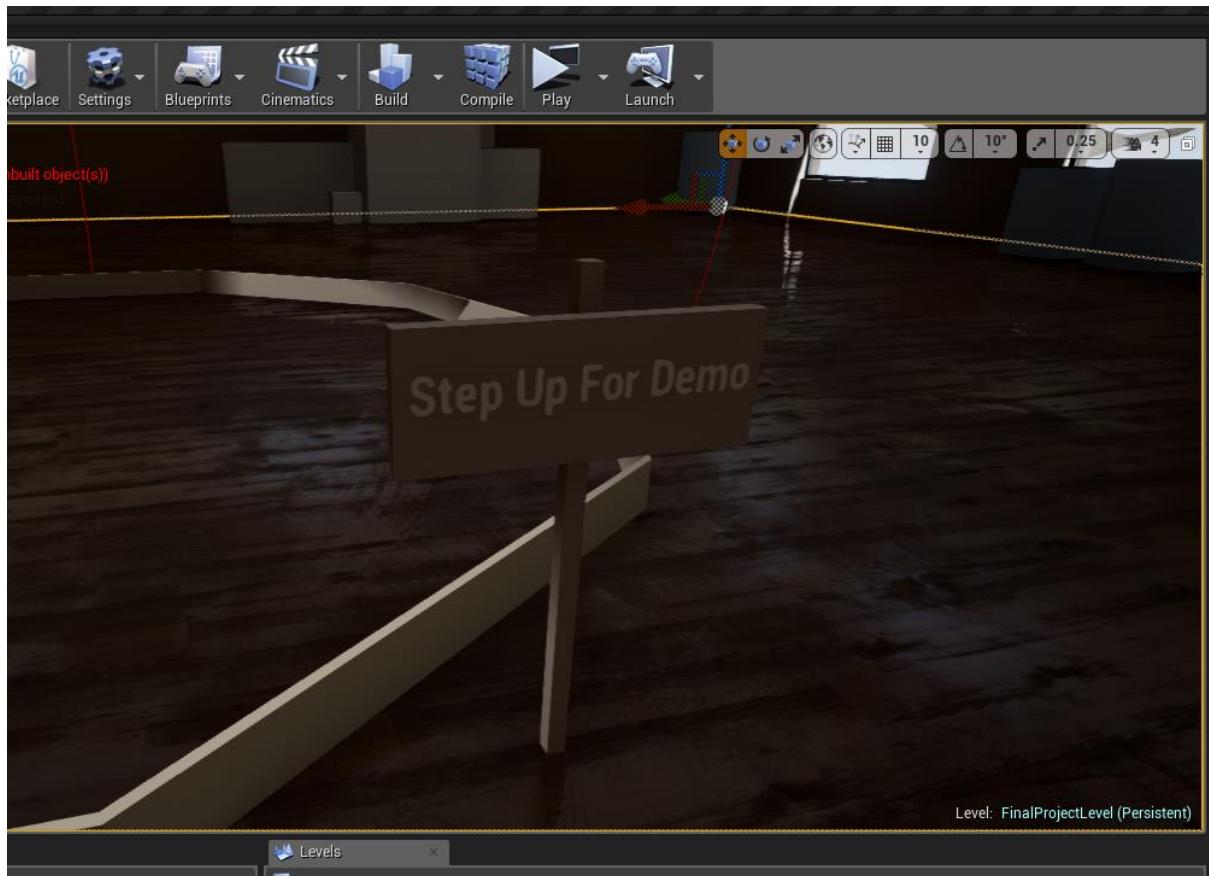


Figure 178

This screenshot shows the LevelSequencePauseTime that had been added to the Actor to allow pauses in the LevelSequencerTimeline in the .h file.

```
22  class FINALPROJECTRYANT_API ACollisionDemo : public AActor
23  {
24      GENERATED_BODY()
25
26  public:
27      // Sets default values for this actor's properties
28      ACollisionDemo();
29
30      // Called when the game starts or when spawned
31      virtual void BeginPlay() override;
32
33      // Called every frame
34      virtual void Tick( float DeltaSeconds ) override;
35
36  private:
37      UPROPERTY(EditAnywhere, Category = "Collision Demo|Verts", Meta = (MakeEditWidget = true))
38          TArray<FVector> LocalSpaceVerts;
39
40      UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
41          FVector StartPoint;
42
43      UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
44          FVector EndPoint;
45
46      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Points")
47          FVector WorldSpaceLineOffset = FVector(0.0f, 0.0f, 0.0f);
48
49      UPROPERTY(EditAnywhere, Category = "Collision Demo|Trigger")
50          USphereComponent* TriggerSphere = nullptr;
51
52      UPROPERTY(EditAnywhere, Category = "Collision Demo|Trigger")
53          ULevelSequence* LevelSequence = nullptr;
54
55      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
56          eCollisionRenderMode CollisionRenderMode = eCollisionRenderMode::FacePlaneWithIntersect;
57
58      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
59          float FacePlaneAlpha = 0.04f;           // 10
60
61      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
62          bool bRenderIntersect = true;
63
64      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Text")
65          float TextContainerIndex = -1;           // -1 default = No Text
66
67      UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Trigger")
68          float LevelSequencePauseTime = 0.0f;
```

Figure 179

This screenshot displays code that had been added to handle pauses in the level sequencer timeline in the .cpp file.

```

61 void ACollisionDemo::Tick( float DeltaTime )
62 {
63     Super::Tick( DeltaTime );
64
65     UpdateWorldSpaceVerts();
66
67     UpdatePlanes();
68
69     PerformCollisionCheck();
70
71     DrawTriangle();
72
73     if ( SequencePlayer )          // Are we playing the level sequence?
74     {
75         if ( LevelSequencePauseTime ) // Sequence is paused?
76         {
77             // If not paused yet
78             if ( !LevelSequencePauseStartTime )
79             {
80                 // Pause it and record the time it was paused
81                 SequencePlayer->Pause();
82                 LevelSequencePauseStartTime = SequencePlayer->GetPlaybackPosition();
83             }
84
85             // Count down the timer
86             LevelSequencePauseTime -= DeltaTime;
87
88             if ( LevelSequencePauseTime <= 0.0f )
89             {
90                 // When timer finished, reset everything and continue playing from old position
91                 LevelSequencePauseTime = 0.0f;
92
93                 SequencePlayer->SetPlaybackPosition(LevelSequencePauseStartTime + 0.1f );      // +0.1 to advance on from pause key
94                 SequencePlayer->StartPlayingNextTick();
95                 LevelSequencePauseStartTime = 0.0f;
96             }
97         }
98     }
99
100    else
101    {
102        // Sequence is not paused so Poll for sequence ending
103        if ( !SequencePlayer->IsPlaying() )
104            OnEndSequnceEvent();           // Clean up sequence player
105    }
106 }

```

Figure 180

This screenshot shows a bool that had been added to CollisionDemoActor to allow the highlighting of the edge planes from the Level Sequencer Timeline in the .h file.

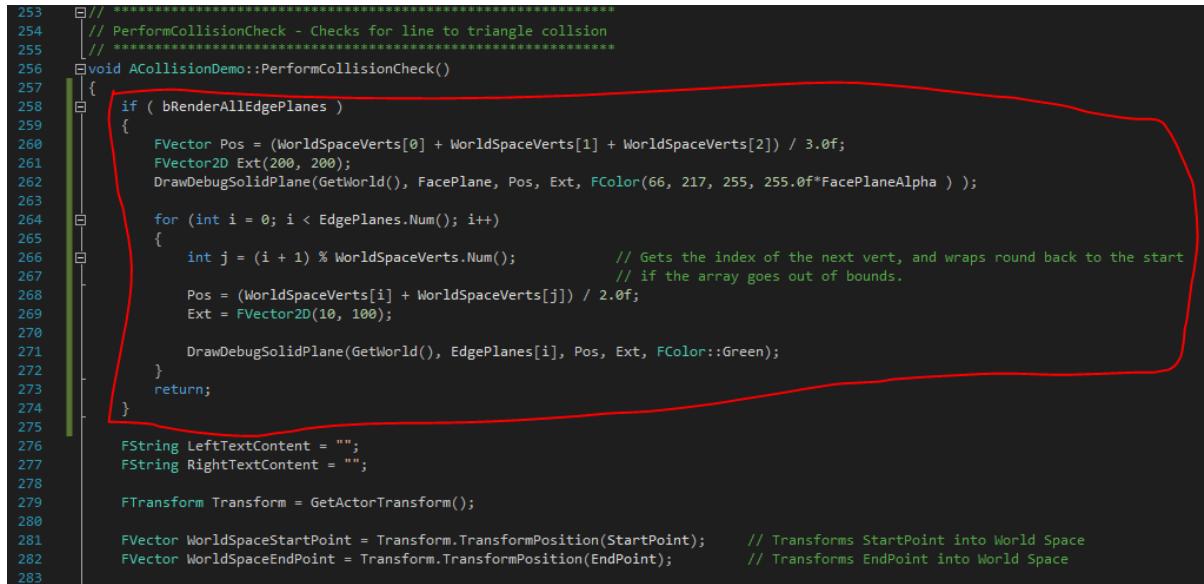
```

61     UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
62     bool bRenderIntersect = true;
63
64     UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Text")
65     float TextContainerIndex = -1;                                // -1 default = No Text
66
67     UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Trigger")
68     float LevelSequencePauseTime = 0.0f;
69
70     float LevelSequencePauseStartTime = 0.0f;
71
72     UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
73     bool bRenderAllEdgePlanes = false;
74

```

Figure 181

The screenshot below shows that code had been added to CollisionDemoActor to allow the highlighting of the edge planes from the Level Sequencer Timeline in the .cpp file.

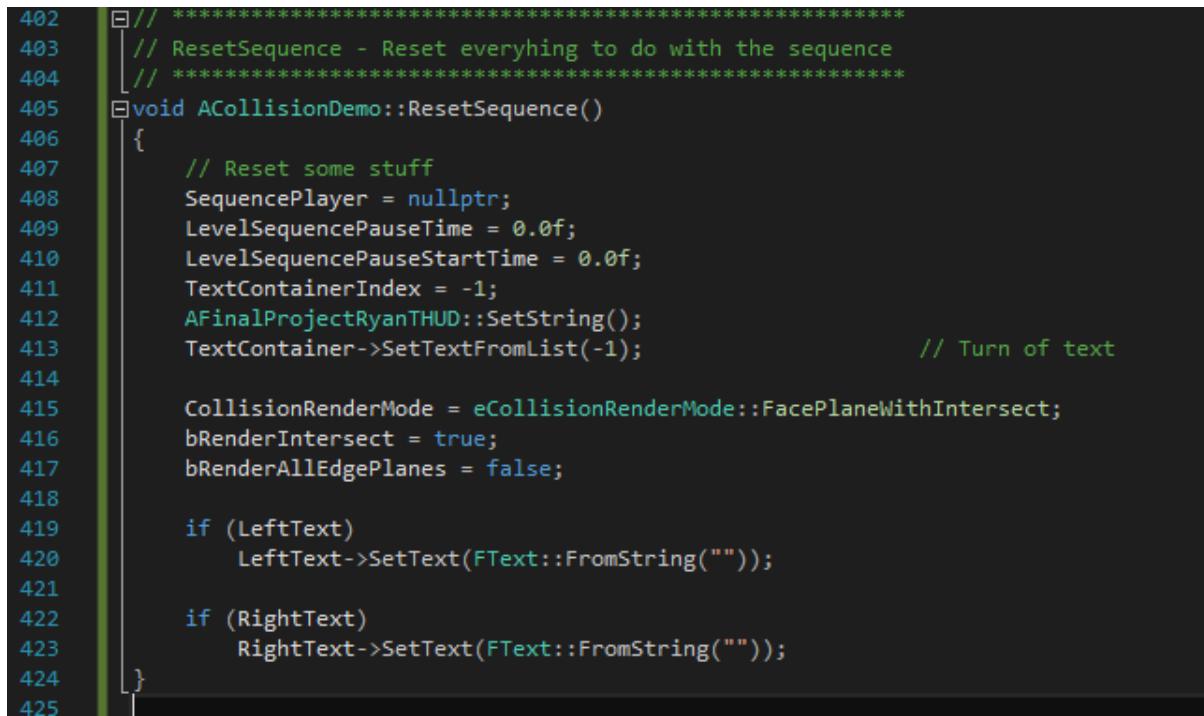


```

253 // ****
254 // PerformCollisionCheck - Checks for line to triangle collision
255 // ****
256 void ACollisionDemo::PerformCollisionCheck()
257 {
258     if ( bRenderAllEdgePlanes )
259     {
260         FVector Pos = (WorldSpaceVerts[0] + WorldSpaceVerts[1] + WorldSpaceVerts[2]) / 3.0f;
261         FVector2D Ext(200, 200);
262         DrawDebugSolidPlane(GetWorld(), FacePlane, Pos, Ext, FColor(66, 217, 255, 255.0f*FacePlaneAlpha) );
263
264         for (int i = 0; i < EdgePlanes.Num(); i++)
265         {
266             int j = (i + 1) % WorldSpaceVerts.Num();           // Gets the index of the next vert, and wraps round back to the start
267                                         // if the array goes out of bounds.
268             Pos = (WorldSpaceVerts[i] + WorldSpaceVerts[j]) / 2.0f;
269             Ext = FVector2D(10, 100);
270
271             DrawDebugSolidPlane(GetWorld(), EdgePlanes[i], Pos, Ext, FColor::Green);
272         }
273     }
274
275     FString LeftTextContent = "";
276     FString RightTextContent = "";
277
278     FTransform Transform = GetActorTransform();
279
280     FVector WorldSpaceStartPoint = Transform.TransformPosition(StartPoint);      // Transforms StartPoint into World Space
281     FVector WorldSpaceEndPoint = Transform.TransformPosition(EndPoint);        // Transforms EndPoint into World Space
282
283 }
```

Figure 182

The below screenshot shows an added reset sequence function in the CollisionDemoActor in the .cpp file.



```

402 // ****
403 // ResetSequence - Reset everything to do with the sequence
404 // ****
405 void ACollisionDemo::ResetSequence()
406 {
407     // Reset some stuff
408     SequencePlayer = nullptr;
409     LevelSequencePauseTime = 0.0f;
410     LevelSequencePauseStartTime = 0.0f;
411     TextContainerIndex = -1;
412     AFinalProjectRyanTHUD::SetString();
413     TextContainer->SetTextFromList(-1);          // Turn of text
414
415     CollisionRenderMode = eCollisionRenderMode::FacePlaneWithIntersect;
416     bRenderIntersect = true;
417     bRenderAllEdgePlanes = false;
418
419     if (LeftText)
420         LeftText->SetText(FText::FromString(""));
421
422     if (RightText)
423         RightText->SetText(FText::FromString(""));
424 }
```

Figure 183

This screenshot shows an added reset sequence function in the CollisionDemoActor in the .h file.

A code editor window showing the CollisionDemoActor.h file. Line 107 contains the function declaration `void ResetSequence();`, which is circled in red.

```
97     private:  
98         void DrawTriangle();  
99         void UpdateWorldSpaceVerts();  
100        void AddVert();  
101        void ClearVerts();  
102        void UpdatePlanes();  
103        void DrawDebugPlanes();  
104        void PerformCollisionCheck();  
105        void OnEndSequenceEvent();  
106        void ResetSequence();  
107
```

Figure 184

This screenshot shows a callback function that had been added so it can trigger when exiting the trigger volume in the .h file.

A code editor window showing the CollisionDemoActor.h file. Line 97 contains the function declaration `void ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex);`, which is circled in red.

```
90     UPROPERTY()  
91     ULevelSequencePlayer* SequencePlayer = nullptr;  
92  
93     UFUNCTION()  
94     void EnterSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);  
95  
96     UFUNCTION()  
97     void ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex);  
98  
99     private:  
100        void DrawTriangle();  
101        void UpdateWorldSpaceVerts();
```

Figure 185

In this screenshot, a callback function had been added so when exiting the trigger volume, the trigger resets the demo sequence in the .cpp file.

A code editor window showing the CollisionDemo.cpp file. Line 128 contains the function implementation `void ACollisionDemo::ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)`. Inside this function, there is a check for overlapping characters and a call to `ResetSequence()`.

```
125     // *****  
126     // ExitSphereVolume - Called when component exits collision volume  
127     // *****  
128     void ACollisionDemo::ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)  
129     {  
130         // only allowed if overlapped by any kind of character  
131         if ( !OtherComp->GetClass()->IsChildOf( ACharacter::StaticClass() ) )  
132             return;  
133  
134         if ( SequencePlayer )  
135         {  
136             SequencePlayer->Stop();  
137             SequencePlayer = nullptr;  
138         }  
139         ResetSequence();  
140  
141         TextContainerIndex = -1;  
142         AFinalProjectRyanTHUD::SetString();  
143     }  
144 }
```

Figure 186

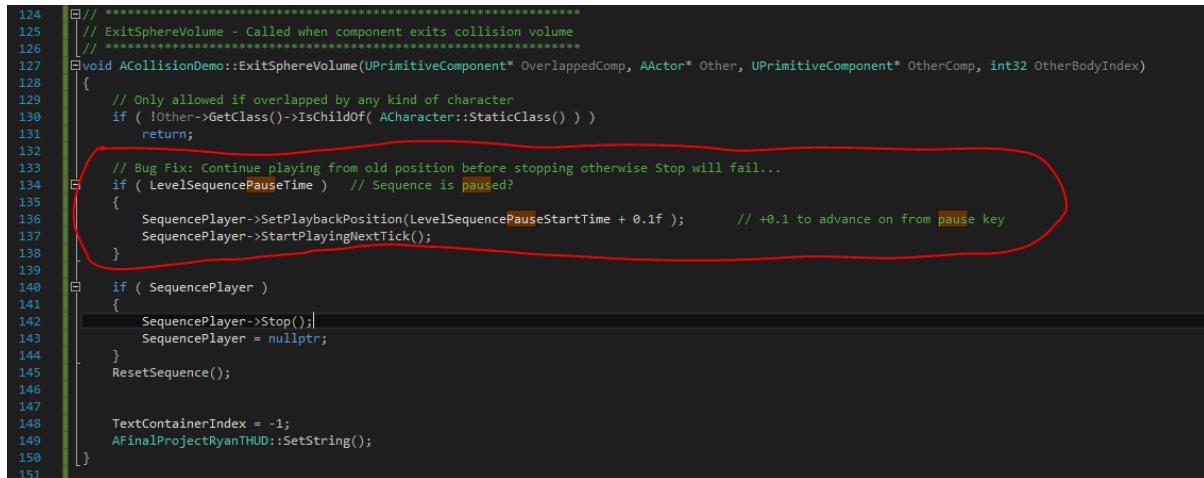
This screenshot displays a callback being registered with the sphere volume using engine call.

A code editor window showing the CollisionDemo.cpp file. Line 29 contains the engine call `TriggerSphere->OnComponentEndOverlap.AddDynamic(this, &ACollisionDemo::ExitSphereVolume);`, which is circled in red.

```
21     // *****  
22     // BeginPlay - Called when the game starts or when spawned  
23     // *****  
24     void ACollisionDemo::BeginPlay()  
25     {  
26         Super::BeginPlay();  
27  
28         TriggerSphere->OnComponentBeginOverlap.AddDynamic(this, &ACollisionDemo::EnterSphereVolume);  
29         TriggerSphere->OnComponentEndOverlap.AddDynamic(this, &ACollisionDemo::ExitSphereVolume);  
30  
31         UpdateWorldSpaceVerts();  
32     }
```

Figure 187

The final screenshot of this section displays a bug being fixed; this bug was contained in the exit volume function and if the sequence was running and the game was paused, it would not stop.



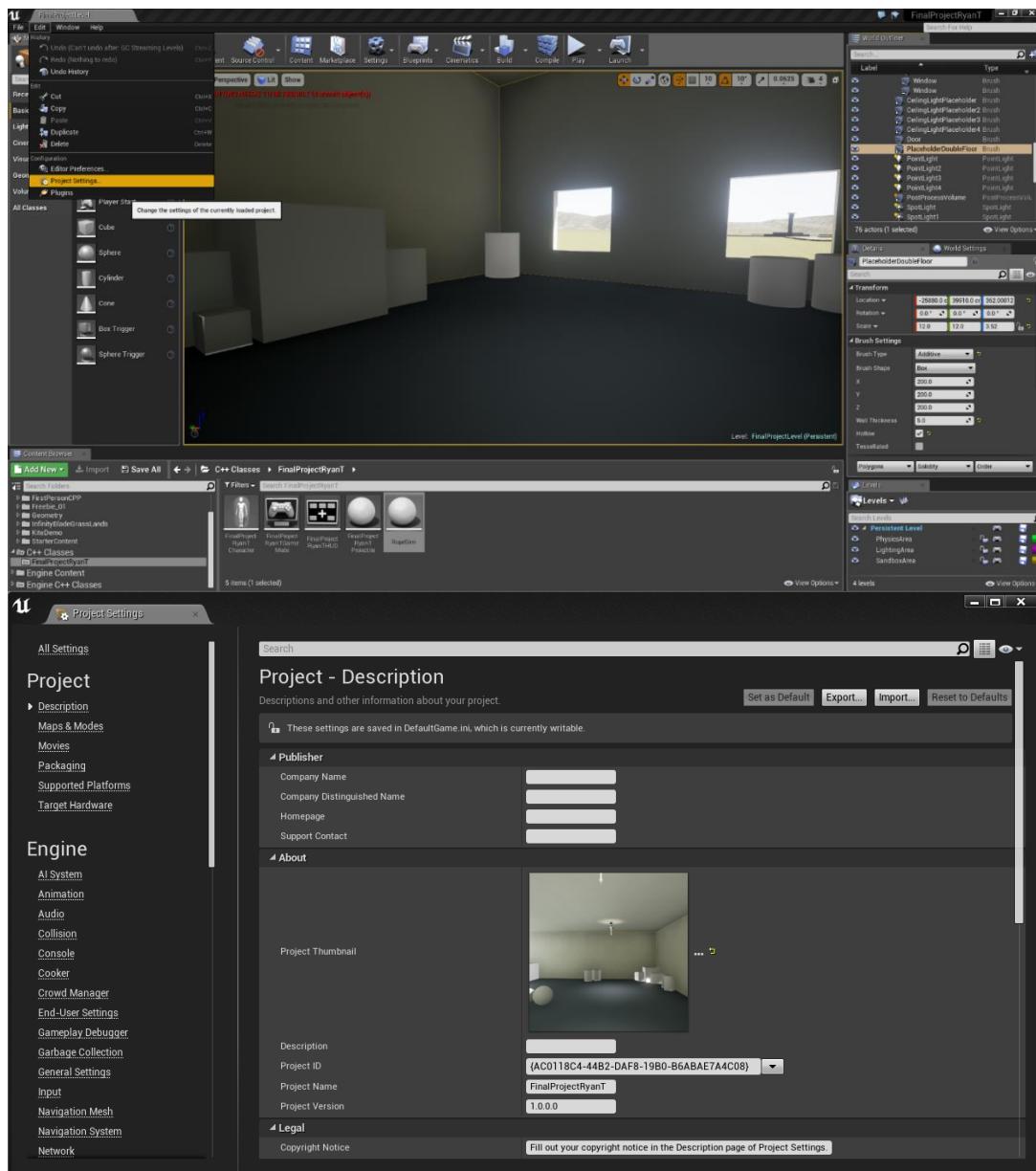
```
124 // ****
125 // ExitSphereVolume - Called when component exits collision volume
126 // ****
127 void ACollisionDemo::ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
128 {
129     // Only allowed if overlapped by any kind of character
130     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
131         return;
132
133     // Bug Fix: Continue playing from old position before stopping otherwise Stop will fail...
134     if ( LevelSequencePauseTime ) // Sequence is paused?
135     {
136         SequencePlayer->SetPlaybackPosition(LevelSequencePauseStartTime + 0.1f); // +0.1 to advance on from pause key
137         SequencePlayer->StartPlayingNextTick();
138     }
139
140     if ( SequencePlayer )
141     {
142         SequencePlayer->Stop();
143         SequencePlayer = nullptr;
144     }
145     ResetSequence();
146
147     TextContainerIndex = -1;
148     AFinalProjectRyanTHUD::SetString();
149 }
150
151
```

A red oval highlights the code block starting at line 134, which contains the bug fix for continuing playback after a pause.

Figure 188

### 6.3. Cloth Simulation Building (Physics Area)

The following three screenshots display the project settings being opened to add a new collision profile for the collision to use.



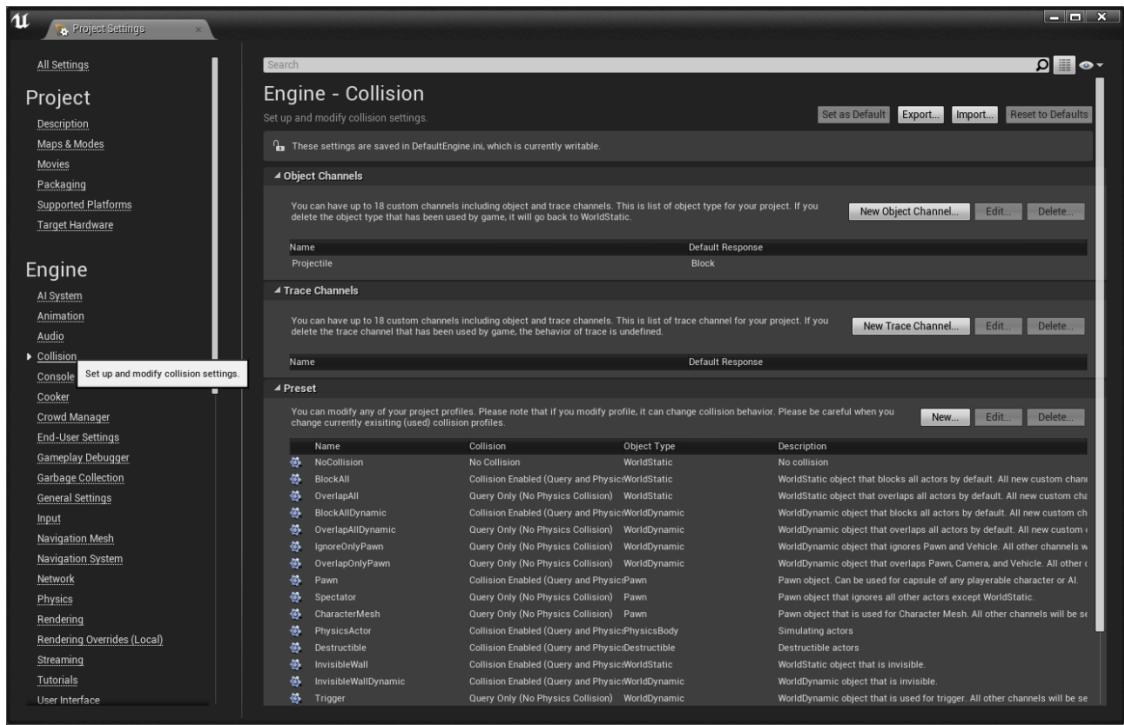


Figure 189 – Project Settings (3 Images)

The two screenshots below are where a new profile was created to allow the rope to fulfil its purpose. The screenshot on the left is the default project settings for a new collision to be set up and the screenshot on the right are the project settings that were used for the rope collision.

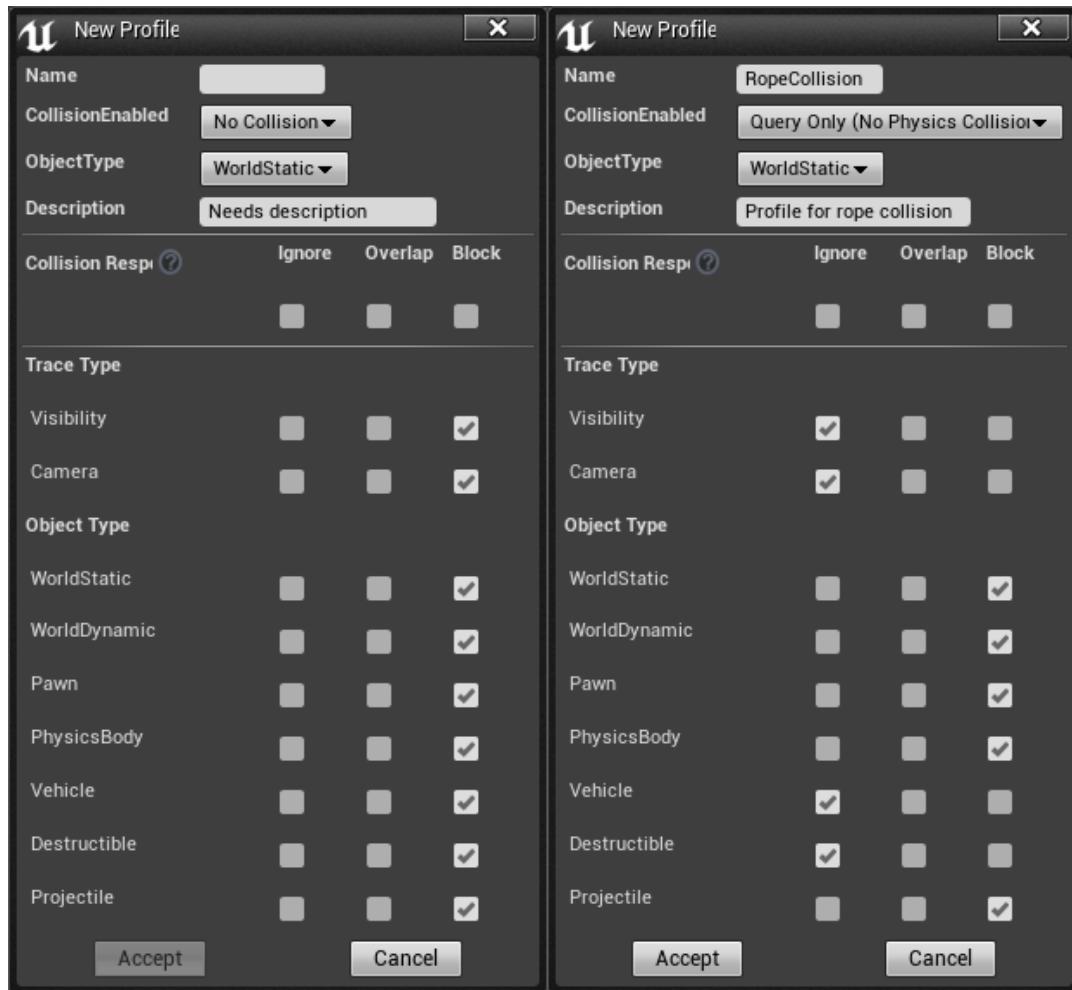


Figure 190

This screenshot shows the new collision profile being highlighted at the bottom of the screen.

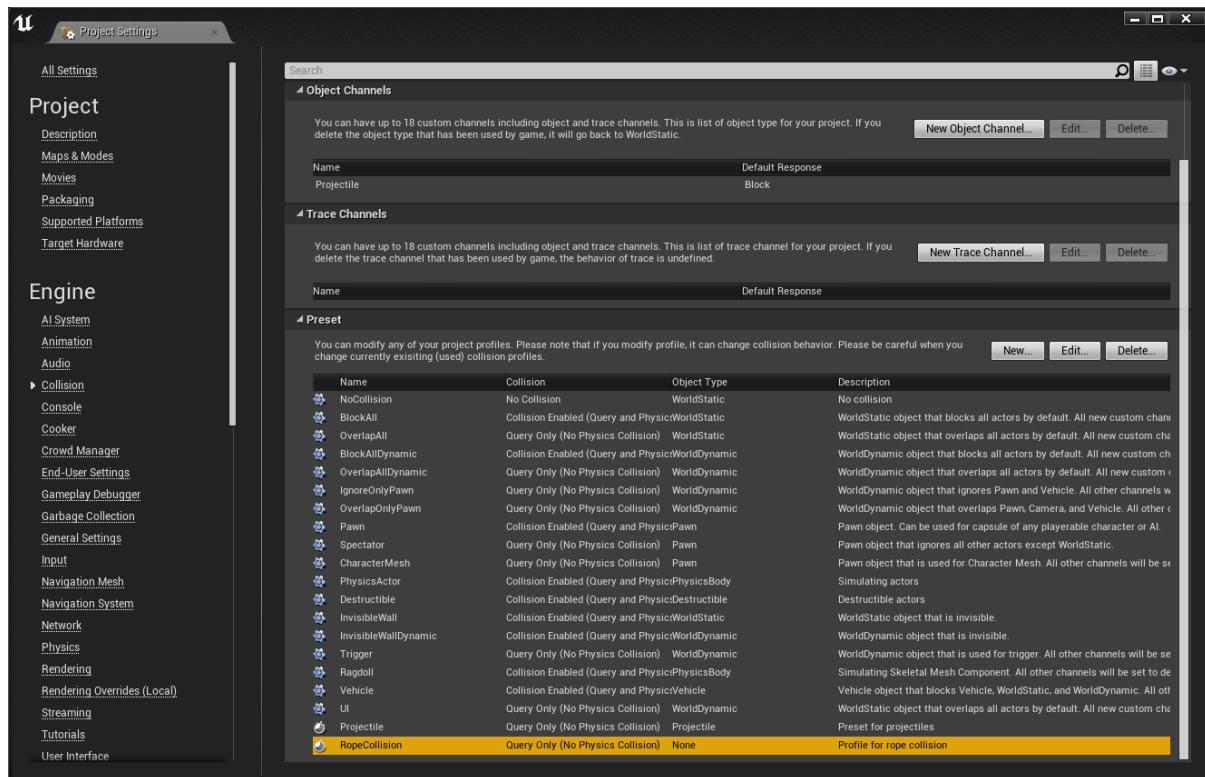


Figure 191

The screenshot below shows a C++ class being created for the rope object in the game.

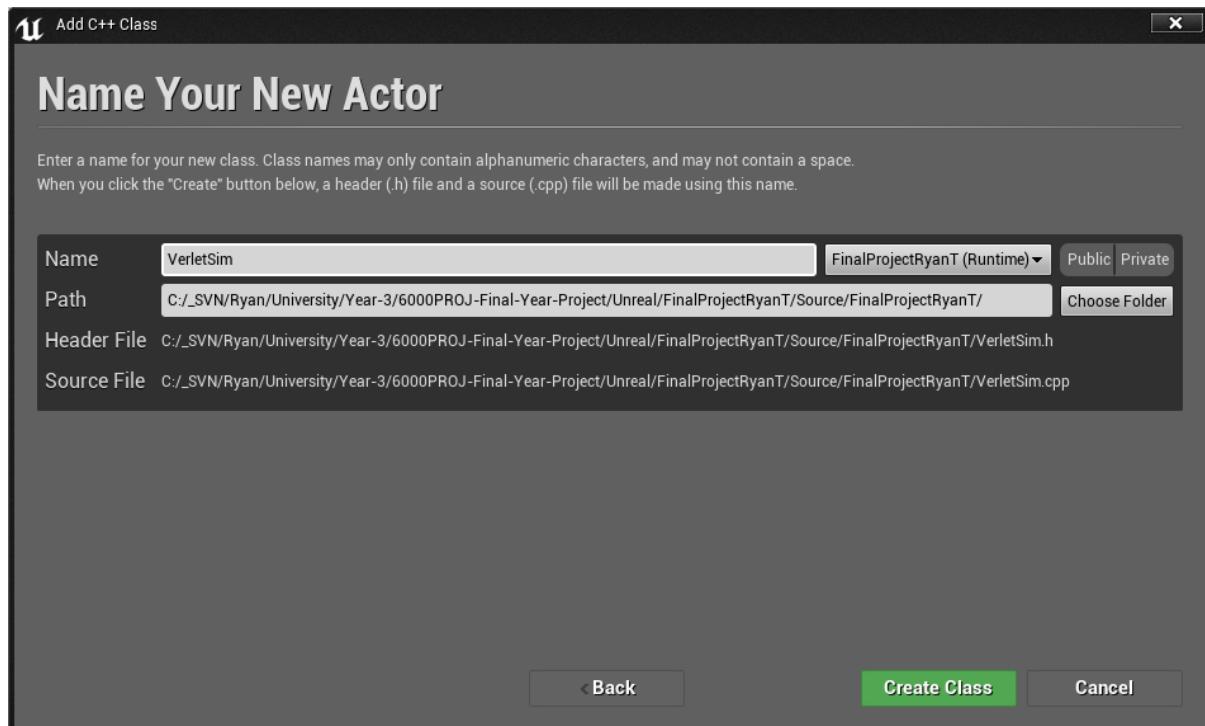
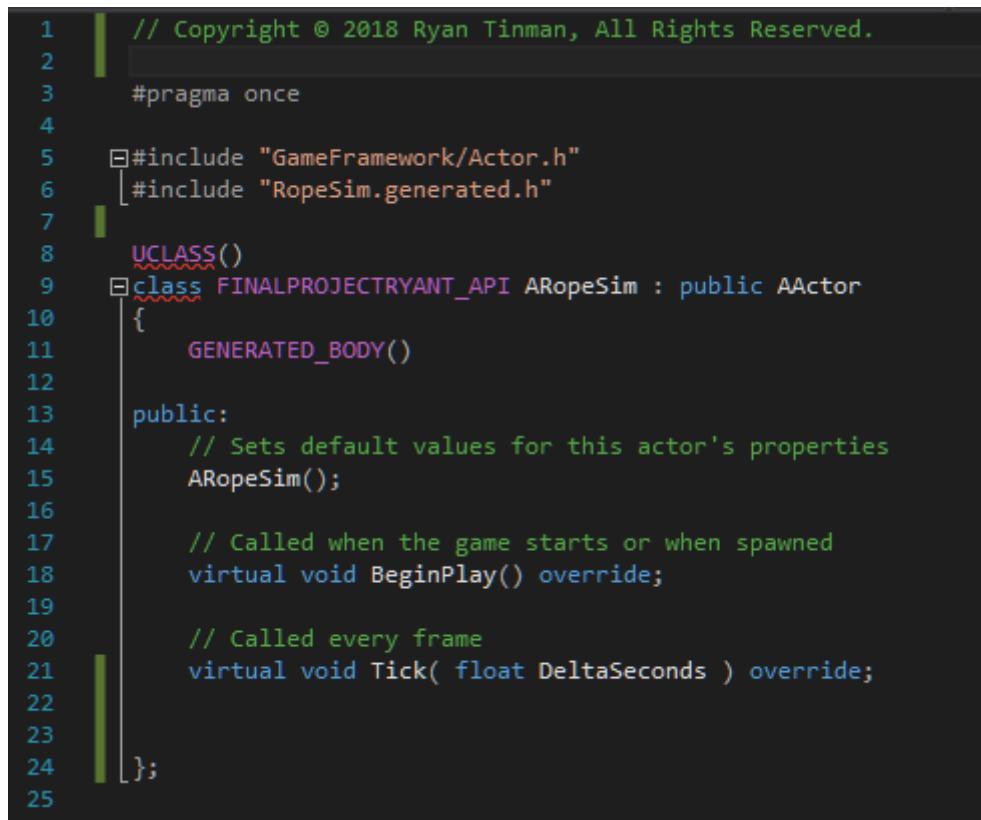


Figure 192

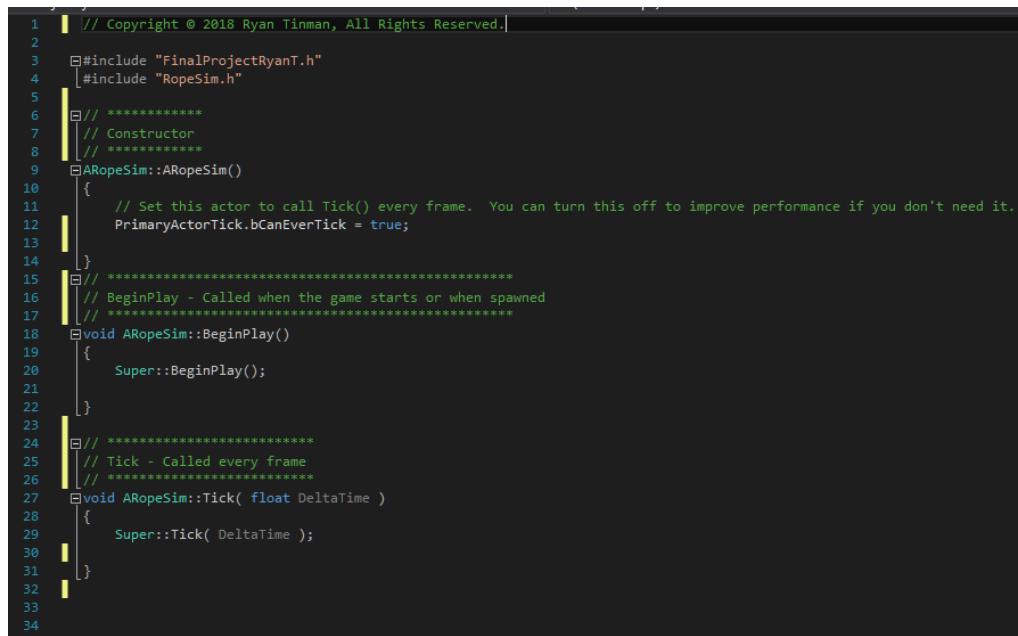
The next screenshot displays the beginning of the initial code for the rope in the .h file.



```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "RopeSim.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API ARopeSim : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     ARopeSim();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23 };
24
25 
```

Figure 193

The next screenshot displays the beginning of the initial code for the rope in the .cpp file.



```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "RopeSim.h"
5
6 // *****
7 // Constructor
8 // *****
9 ARopeSim::ARopeSim()
10 {
11     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
12     PrimaryActorTick.bCanEverTick = true;
13 }
14
15 // *****
16 // BeginPlay - Called when the game starts or when spawned
17 // *****
18 void ARopeSim::BeginPlay()
19 {
20     Super::BeginPlay();
21 }
22
23 // *****
24 // Tick - Called every frame
25 // *****
26 void ARopeSim::Tick( float DeltaTime )
27 {
28     Super::Tick( DeltaTime );
29 }
30
31
32
33
34 
```

Figure 194

This screenshot shows the values being added to the .h file in preparation for the code.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "RopeSim.generated.h"
7
8 struct RopeVert
9 {
10     unsigned int Flags;
11     FVector Vert;
12 };
13
14 UCLASS()
15 class FINALPROJECTRYANT_API ARopeSim : public AActor
16 {
17     GENERATED_BODY()
18
19 public:
20     // Sets default values for this actor's properties
21     ARopeSim();
22
23     // Called when the game starts or when spawned
24     virtual void BeginPlay() override;
25
26     // Called every frame
27     virtual void Tick( float DeltaSeconds ) override;
28
29
30 private:
31     void Constrain();
32     void Simulate(float DeltaTime);
33     void Render();
34
35     static const int NumVerts = 50;
36
37     float ConstraintDist = 10.0f;
38
39     RopeVert Verts[NumVerts];
40 };
41
```

Figure 195

This screenshot shows the first functions of the code in the .cpp file; this includes setting up the Constructor, BeginPlay and Tick.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "RopeSim.h"
5 #include "DrawDebugHelpers.h"
6
7 // *****
8 // Constructor
9 // *****
10 ARopeSim::ARopeSim()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14
15     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
16
17     FVector Pos = GetActorLocation();
18     for (int i = 0; i < NumVerts; i++)
19     {
20         Verts[i].Flags = 0;
21         Verts[i].Vert = Pos;
22         Pos.Z -= ConstraintDist;
23     }
24 }
25
26 // *****
27 // BeginPlay - Called when the game starts or when spawned
28 // *****
29 void ARopeSim::BeginPlay()
30 {
31     Super::BeginPlay();
32 }
33
34 // *****
35 // Tick - Called every frame
36 // *****
37 void ARopeSim::Tick( float DeltaTime )
38 {
39     Super::Tick( DeltaTime );
40
41     Verts[0].Vert = GetActorLocation();
42
43     Simulate(DeltaTime);
44
45     Constrain();
46
47     Render();
48 }
49
50 }
```

Figure 196

The next screenshot shows the next functions of the code in the .cpp file; this includes setting up the Constrain, Simulate and Render.

```
50
51     // ****
52     // Constrain - Maintain equal distance between all verts
53     // ****
54     void ARopeSim::Constrain()
55     {
56         for (int i = 0; i < NumVerts - 1; i++)
57         {
58             FVector CurrentVert = Verts[i].Vert;
59
60             FVector NextVert = Verts[i+1].Vert;
61
62             FVector Direction = NextVert - CurrentVert;
63
64             Direction.Normalize();
65
66             Verts[i + 1].Vert = CurrentVert + Direction * ConstraintDist;
67         }
68
69         // TODO: Add collision constraints
70     }
71
72     // ****
73     // Simulate - Simulate verlet physics for rope
74     // ****
75     void ARopeSim::Simulate(float DeltaTime)
76     {
77         for (int i = 0; i < NumVerts; i++)
78         {
79             // TODO: Add force due to gravity
80             Verts[i].Vert.Z += -980.0f * DeltaTime;
81
82             // TODO: Add force due to wind
83         }
84     }
85
86     // ****
87     // Render - Renders the rope
88     // ****
89     void ARopeSim::Render()
90     {
91         for (int i = 0; i < NumVerts - 1; i++)
92         {
93             DrawDebugLine(GetWorld(), Verts[i].Vert, Verts[i + 1].Vert, (i&1) ? FColor::Red : FColor::Blue, false, -1.0f, 0, 5.0f);
94         }
95     }
96
97
98
```

Figure 197

This screenshot displays the OnConstruction function being added to set flags in the .cpp file.

```
22
23     void ARopeSim::OnConstruction(const FTransform& Transform)
24     {
25         Super::OnConstruction(Transform);
26
27         FVector Pos = Transform.GetLocation();
28         for (int i = 0; i < NumVerts; i++)
29         {
30             Verts[i].Flags = 0;
31             Verts[i].Pos = Pos;
32             Verts[i].PreviousPos = Pos;
33             Pos.Z -= ConstraintDist;
34         }
35     }
36
```

Figure 198

The screenshot below shows the rope class appearing in unreal. To add it to the scene, the object just has to be dragged and dropped into the scene view.

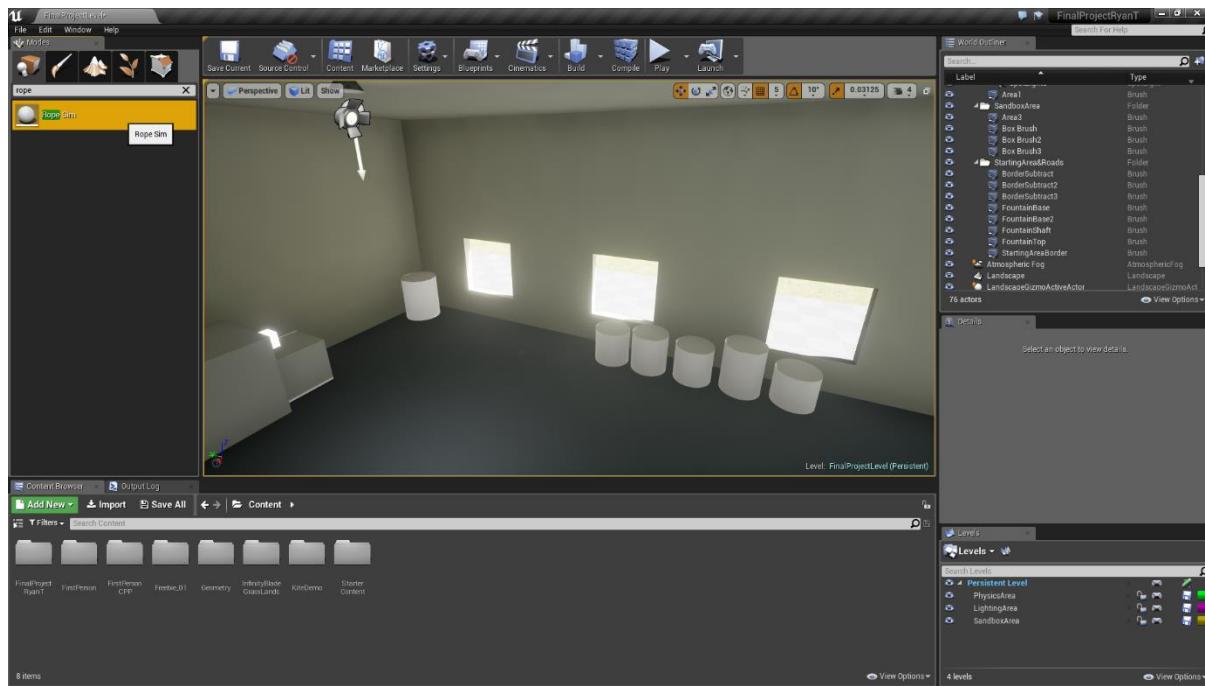


Figure 199

The screenshot below shows that the rope object is now in the scene.



Figure 200

The screenshot shows that the rope is now correctly displaying in the world as the verts are constrained together. The screenshot shows the rope is being simulated and rendered as debug lines.



Figure 201

This screenshot shows UProperties being added to the .h file. This allows values to be exposed to the unreal editor, allowing changes at runtime and while editing without changing code.

```
34 // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
35 // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.
36
37 // Toggles display of debug draw lines
38 UPROPERTY(EditAnywhere, Category = "Rope Settings", meta = (DisplayName = "Enable Debug Draw"))
39 bool bEnableDebugDraw = false;
40
41 // Mass for each vertex of the rope
42 UPROPERTY(EditAnywhere, Category = "Rope Settings", meta = (ClampMin = 0.0f))
43 float ParticleMass = 5.0f;
44
45 // The amount of damping to be imparted on the rope
46 UPROPERTY(EditAnywhere, Category = "Rope Settings", meta = (ClampMin = 0.0f, DisplayName = "Damping Amount"))
47 float DampingAmt = 0.01f;
48
49
```

Figure 202

The next three screenshots show a new class being added, derived and named in Unreal.

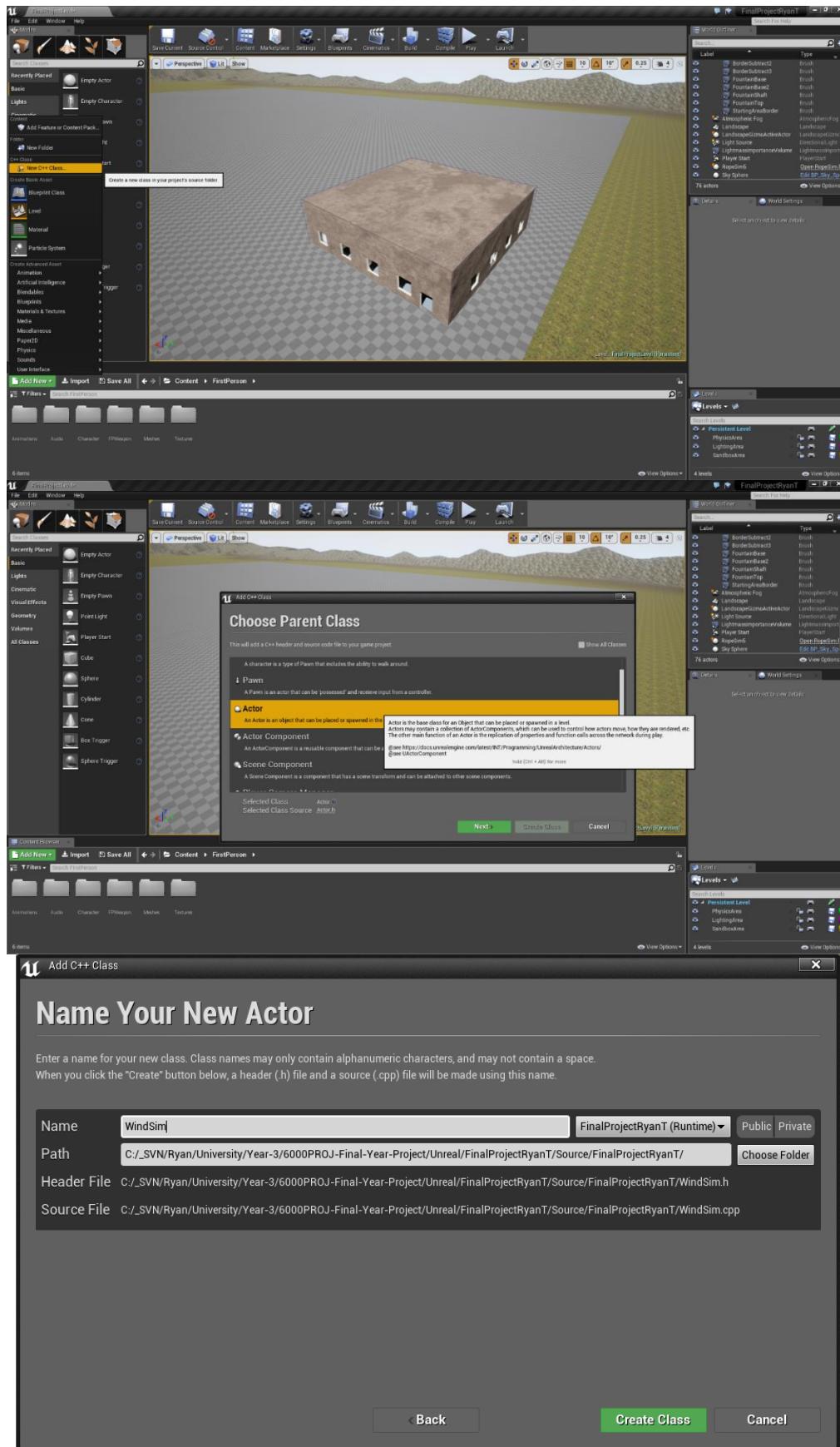


Figure 203 – Adding a New Class (Three Images)

This screenshot shows the wind class being made and code being added to it in the .h file.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 [ #include "WindSim.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API AWindSim : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     AWindSim();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23 };
24
25
26
```

Figure 204

This screenshot shows the initial code for the wind being implemented into the .cpp file. There was a scene component added to give the wind object a transform.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 [ #include "WindSim.h"
5
6 // ****
7 // Constructor
8 // ****
9 AWindSim::AWindSim()
10 {
11     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
12     PrimaryActorTick.bCanEverTick = true;
13
14     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
15     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
16 }
17
18 // ****
19 // BeginPlay - Called when the game starts or when spawned
20 // ****
21 void AWindSim::BeginPlay()
22 {
23     Super::BeginPlay();
24 }
25
26 // ****
27 // Tick - Called every frame
28 // ****
29 void AWindSim::Tick( float DeltaTime )
30 {
31     Super::Tick( DeltaTime );
32 }
33
34
35
```

Figure 205

This screenshot shows the values being added into the .h file in preparation for coding.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "WindSim.h"
5 #include "DrawDebugHelpers.h"
6
7 // *****
8 // Constructor
9 // *****
10 AWindSim::AWindSim()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14
15     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
16     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
17 }
18
19 // *****
20 // BeginPlay - Called when the game starts or when spawned
21 // *****
22 void AWindSim::BeginPlay()
23 {
24     Super::BeginPlay();
25 }
26
27 // *****
28 // Tick - Called every frame
29 // *****
30 void AWindSim::Tick( float DeltaTime )
31 {
32     Super::Tick( DeltaTime );
33
34     Time += DeltaTime;
35
36     // Listener for debug draw tickbox
37     if(bEnableDebugDraw)
38         DebugDraw();
39 }
40
```

Figure 206

This screenshot shows the first stages of coding for the wind simulation in the .cpp file; the first three variables consist of Constructor, BeginPlay and Tick.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "WindSim.h"
5 #include "DrawDebugHelpers.h"
6
7 // *****
8 // Constructor
9 // *****
10 AWindSim::AWindSim()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14
15     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
16     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
17 }
18
19 // *****
20 // BeginPlay - Called when the game starts or when spawned
21 // *****
22 void AWindSim::BeginPlay()
23 {
24     Super::BeginPlay();
25 }
26
27 // *****
28 // Tick - Called every frame
29 // *****
30 void AWindSim::Tick( float DeltaTime )
31 {
32     Super::Tick( DeltaTime );
33
34     Time += DeltaTime;
35
36     // Listener for debug draw tickbox
37     if(bEnableDebugDraw)
38         DebugDraw();
39 }
40
```

Figure 207

This screenshot shows the first stages of coding for the wind simulation in the .cpp file; the next three variables in the code consist of GetWindForce, GetMultiplierAtLocation and DebugDraw.

```

40 // ****
41 // GetWindForce - Returns the force of the wind
42 // ****
43 FVector AWindSim::GetWindForce(FVector WorldPosition)
44 {
45     FVector WindPos = GetActorLocation();
46
47     FVector Force = WorldPosition - WindPos;           // Get the vector pointing from the WindPos towards the WorldPosition
48
49     Force.Normalize();                                // Normalises the force vector, so it is only one world unit long
50
51     FVector WindMultiplier = GetWindMultiplierAtLocation(WorldPosition); // TODO: COMMENT
52
53     return Force * WindMultiplier * Strength;        // Returns the Force multiplied by WindMultiplier multiplied by Strength
54 }
55
56 // ****
57 // GetWindMultiplierAtLocation - Desc...
58 // ****
59 FVector AWindSim::GetWindMultiplierAtLocation(FVector WorldPos)
60 {
61     float T = Time * GustFrequency;                  // Phase multiplier (Speed)
62     float ModX = WorldPos.X * GustXWavelength;    // Wavelength X
63     float ModY = WorldPos.Y * GustYWavelength;    // Wavelength Y
64     float ModZ = WorldPos.Z * GustZWavelength;    // Wavelength Z
65
66     ModX = (1.0f + sinf(ModX + T))*0.5f;          // Get sine value (-1 to +1) and convert into range 0 to 1
67     ModY = (1.0f + sinf(ModY + T))*0.5f;
68     ModZ = (1.0f + sinf(ModZ + T))*0.5f;
69
70     // Return values are from 0.0 to 1.0 and are used to scale the wind direction
71
72     return FVector(ModX, ModY, ModZ);
73 }
74
75 // ****
76 // DebugDraw - Debug display for wind
77 // ****
78 void AWindSim::DebugDraw()
79 {
80     FVector Centre = GetActorLocation();
81
82     for (float X = -200; X < 200; X += 40)
83     {
84         for (float Y = -200; Y < 200; Y += 40)
85         {
86             for (float Z = -200; Z < 200; Z += 40)
87             {
88                 FVector Pos = Centre + FVector(X, Y, Z);
89
90                 FVector Force = GetWindForce(Pos);
91
92                 DrawDebugLine(GetWorld(), Pos, Pos + Force, FColor::Green, false, -1.0f, 0, 3.0f);
93             }
94         }
95     }
96 }
```

Figure 208

This screenshot shows the collision building being duplicated to make the rope and the cloth building.

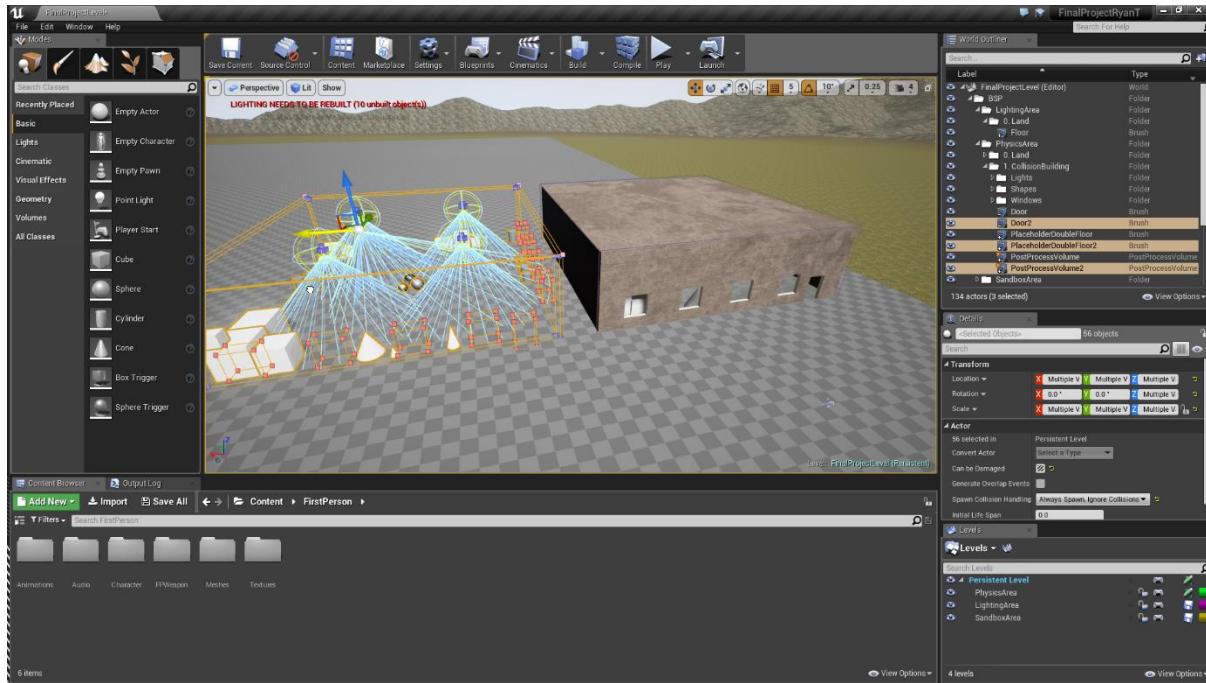


Figure 209

This screenshot shows the completed empty rope and cloth building.



Figure 210

This screenshot shows the rope being moved from the collision building into the new building.



Figure 211

This screenshot displays the first rope being duplicated multiple times so that more than one can be in the scene.

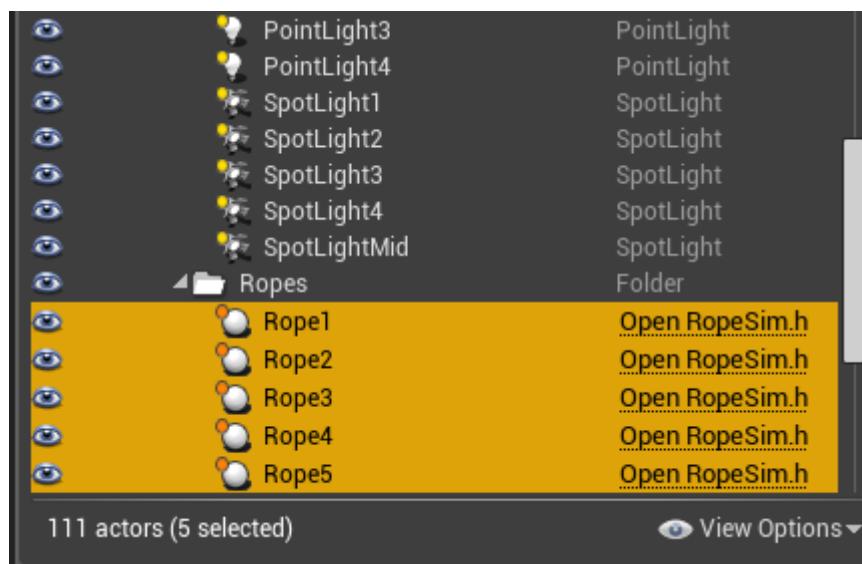


Figure 212

The screenshot below shows geometry that was added into the scene for the ropes to interact with.



Figure 213

This screenshot shows the wind class appearing in unreal. It was then added into the scene using the drag and drop system unreal has.

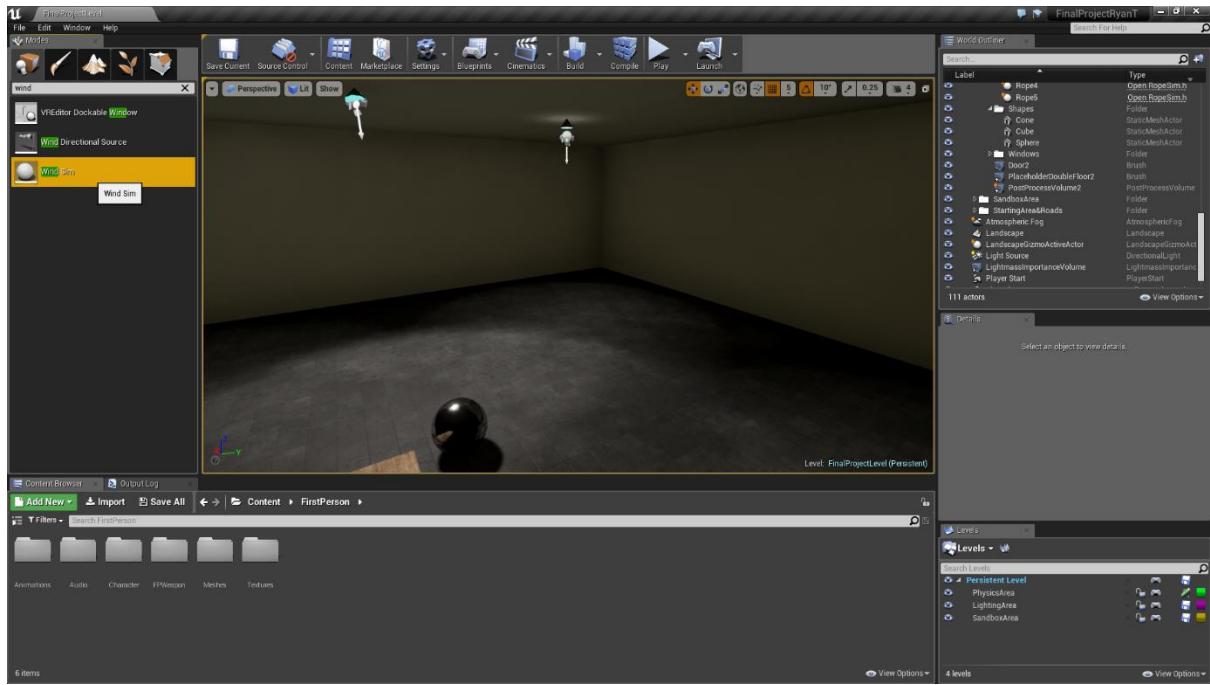


Figure 214

This screenshot shows the gameplay screen looking at the wind object with debug lines on it.

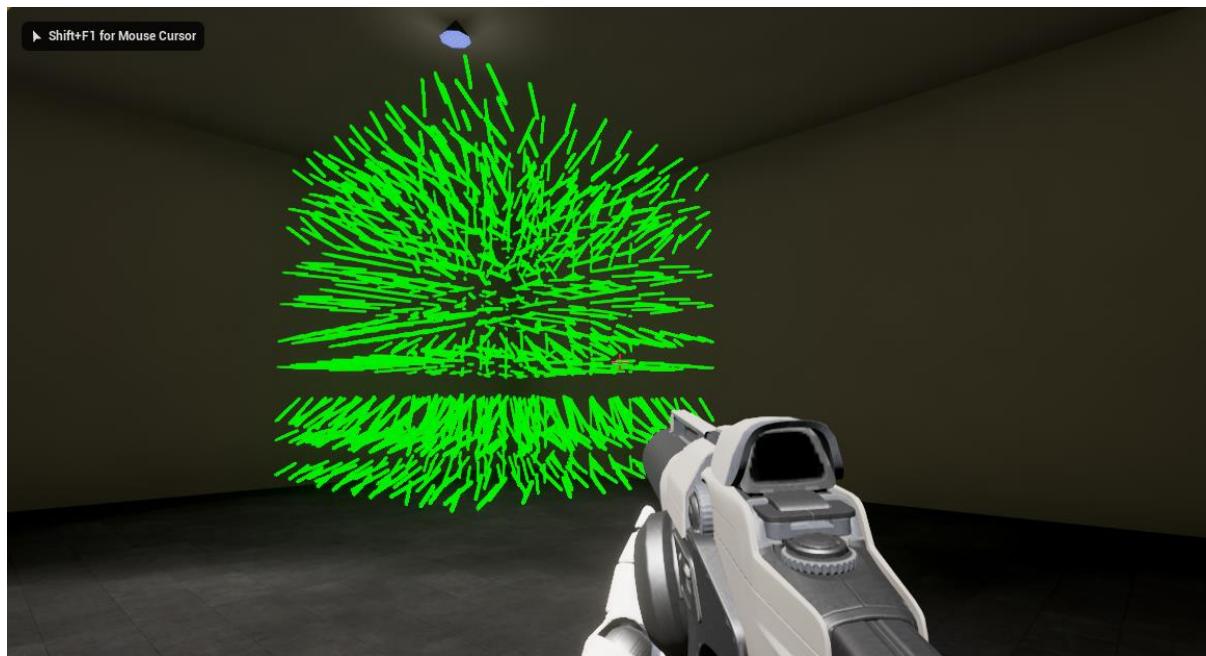


Figure 215

The screenshot below shows the VerletSim class being made in the .h file. This will contain functionality for the rope and cloth; they will derive from VerletSim rather than Actor.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "VerletSim.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API AVerletSim : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     AVerletSim();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23 };
```

Figure 216

This screenshot shows the initial code for the VerletSim .cpp file. The USceneComponent has been added to this file.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "VerletSim.h"
5
6
7 // *****
8 // Constructor
9 // *****
10 AVerletSim::AVerletSim()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14
15     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
16     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
17 }
18
19 // *****
20 // BeginPlay - Called when the game starts or when spawned
21 // *****
22 void AVerletSim::BeginPlay()
23 {
24     Super::BeginPlay();
25 }
26
27
28 // *****
29 // Tick - Called every frame
30 // *****
31 void AVerletSim::Tick( float DeltaTime )
32 {
33     Super::Tick( DeltaTime );
34 }
35
36
```

Figure 217

This screenshot shows the WindSim being included in the .h file so it has access to the wind class. This code also moves the structure from the rope to VerletSim so both rope and cloth can use it.

```
4
5 #include "GameFramework/Actor.h"
6 #include "WindSim.h"
7 #include "VerletSim.generated.h"
8
9 // Structure that represents a single vertex
10 struct RopeVert
11 {
12     unsigned int Flags;
13     FVector Pos;
14     FVector PreviousPos;
15 };
16
```

Figure 218

This screenshot shows values being added to the .h file in preparation for the code.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "WindSim.h"
7 #include "VerletSim.generated.h"
8
9 // Structure that represents a single vertex
10 struct RopeVert
11 {
12     unsigned int Flags;
13     FVector Pos;
14     FVector PreviousPos;
15 };
16
17 UCLASS()
18 class FINALPROJECTRYANT_API AVerletSim : public AActor
19 {
20     GENERATED_BODY()
21
22 public:
23     // Sets default values for this actor's properties
24     AVerletSim();
25
26     // Called when the game starts or when spawned
27     virtual void BeginPlay() override;
28
29     // Called every frame
30     virtual void Tick( float DeltaSeconds ) override;
31
32 protected:
33     // Mass for each vertex of the rope
34     UPROPERTY(EditAnywhere, Category = "Rope Settings", meta = (ClampMin = 0.0f))
35         float ParticleMass = 5.0f;
36
37     // The amount of damping to be imparted on the rope
38     UPROPERTY(EditAnywhere, Category = "Rope Settings", meta = (ClampMin = 0.0f, DisplayName = "Damping Amount"))
39         float DampingAmt = 0.01f;
40
41     AWindSim* Wind = nullptr;
42
43     float ConstraintDist = 10.0f;
44
45 protected:
46     void SimulateVert(RopeVert &Vert, float DeltaTime);
47
48     bool ResolveVertCollision(RopeVert &Vert);
49
50     void ConstrainVert(const RopeVert &Vert, RopeVert &VertToConstrain);
51 };
52
```

Figure 219

The next two screenshots show the first stages of coding in the VerletSim.cpp file; the functions included are Constructor, BeginPlay, Tick, SimulateVert, ResolveVertCollision and ConstrainVert.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "VerletSim.h"
5
6
7 // ****
8 // Constructor
9 // ****
10 AVerletSim::AVerletSim()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14
15     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
16     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
17 }
18
19 // ****
20 // BeginPlay - Called when the game starts or when spawned
21 // ****
22 void AVerletSim::BeginPlay()
23 {
24     Super::BeginPlay();
25
26     // Finds the first wind source object in the level
27     if (!Wind)
28     {
29         // TODO: COMMENT
30         TArray<AActor*> WindList;
31         UGameplayStatics::GetAllActorsOfClass(GetWorld(), AWindSim::StaticClass(), WindList);
32
33         if (WindList.Num())
34             Wind = Cast<AWindSim>(WindList[0]);
35     }
36 }
37
38 // ****
39 // Tick - Called every frame
40 // ****
41 void AVerletSim::Tick( float DeltaTime )
42 {
43     Super::Tick( DeltaTime );
44 }
```

Figure 220 – VerletSim (Part 1)

```

47 // ****
48 // SimulateVert - Handles the simulation of verts
49 // ****
50
51 void AVerletSim::SimulateVert(RopeVert &Vert, float DeltaTime)
52 {
53     Vert.PreviousPos = Vert.Pos;
54
55     // If disable simulation flag is set, return
56     if (Vert.Flags)
57         return;
58
59     // Zero force to begin with
60     FVector Force(0.0f, 0.0f, 0.0f);
61
62     // Add gravity force
63     Force += FVector(0.0f, 0.0f, -980.0f);      // TODO: Pass in UE4 gravity setting instead of hardcoding
64
65     // Add wind force
66     if (Wind)
67     {
68         // Get the wind force for the world position of the current vertex.
69         Force += Wind->GetWindForce(Vert.Pos);
70     }
71
72     // Dividing force by particle mass
73     if (ParticleMass != 0.0f)          // Avoid divide by 0
74         Force /= ParticleMass;
75
76     Vert.Pos += (Vert.Pos - Vert.PreviousPos) * DampingAmt;
77
78     // Add the sum of all forces to the current vert, multiplied by DeltaTime
79     Vert.Pos += Force * DeltaTime;
80 }
81
82 // ****
83 // ResolveVertCollision - Resolves the collision for verts
84 // ****
85 bool AVerletSim::ResolveVertCollision(RopeVert &Vert)
86 {
87     FHitResult HitResult;
88     FCollisionQueryParams Params;
89
90     if (GetWorld()->LineTraceSingleByProfile(HitResult, Vert.PreviousPos, Vert.Pos, FName("RopeCollision"), Params))
91     {
92         Vert.Pos = HitResult.ImpactPoint;
93
94         return true;
95     }
96
97     return false;
98 }
99
100 // ****
101 // ConstrainVert - Constrains the verts
102 // ****
103 void AVerletSim::ConstrainVert(const RopeVert &Vert, RopeVert &VertToConstrain)
104 {
105     FVector CurrentVert = Vert.Pos;
106
107     FVector NextVert = VertToConstrain.Pos;
108
109     FVector Direction = NextVert - CurrentVert;
110
111     Direction.Normalize();
112
113     VertToConstrain.Pos = CurrentVert + Direction * ConstraintDist;
114 }

```

Figure 221 – VerletSim (Part 2)

The two screenshots below show the class being changed to derive from the newly created Verlet class rather than the Actor.

```
16
17     UCLASS()
18     =class FINALPROJECTRYANT_API ARopeSim : public AActor|
19     {
```

Figure 222

```
/
8     UCLASS()
9     =class FINALPROJECTRYANT_API ARopeSim : public AVerletSim
10    {
11        GENERATED_BODY()
```

Figure 223

Now that the core functionality has been taken from RopeSim and put into VerletSim, the next two screenshots are what the new RopeSim .cpp file looks like.

```
2
3     #include "FinalProjectRyanT.h"
4     #include "RopeSim.h"
5     #include "DrawDebugHelpers.h"
6     #include "Kismet/GameplayStatics.h"
7     #include "SceneManagement.h"
8
9     // *****
10    // Constructor
11    // *****
12    ARopeSim::ARopeSim()
13    {
14        // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
15        PrimaryActorTick.bCanEverTick = true;
16    }
17
18    // *****
19    // BeginPlay - Called when the game starts or when spawned
20    // *****
21    void ARopeSim::BeginPlay()
22    {
23        Super::BeginPlay();
24    }
25
26    // *****
27    // OnConstruction - Called when instance of this class is spawned or placed in editor
28    // *****
29    void ARopeSim::OnConstruction(const FTransform& Transform)
30    {
31        Super::OnConstruction(Transform);
32
33        // Initialising all verts in rope
34        FVector Pos = Transform.GetLocation();
35
36        for (int i = 0; i < NumVerts; i++)
37        {
38            Verts[i].Flags = 0;
39            Verts[i].Pos = Pos;
40            Verts[i].PreviousPos = Pos;
41            Pos.Z -= ConstraintDist;
42        }
43    }
44
45    // *****
46    // Tick - Called every frame
47    // *****
48    void ARopeSim::Tick( float DeltaTime )
49    {
50        Super::Tick( DeltaTime );
51
52        // Locks first vert of the rope to the actor transform location and disables simulation
53        Verts[0].Pos = GetActorLocation();
54        Verts[0].Flags = 1;
55
56        Simulate(DeltaTime);
57
58        Constrain();
59
60        Render();
61    }
62
```

Figure 224 – RopeSim (Part 1)

```

62
63     // ****
64     // Constrain - Maintain equal distance between all verts
65     // ****
66     void ARopeSim::Constrain()
67     {
68         for (int i = 0; i < NumVerts; i++)
69         {
70             // Resolve collision for current vert
71             if (ResolveVertCollision(Verts[i]))
72             {
73                 // Listener for rope debug draw tickbox
74                 if (bEnableDebugDraw)
75                     DebugDraw(i);
76             }
77         }
78
79         for (int i = 0; i < NumVerts - 1; i++)
80             ConstrainVert(Verts[i], Verts[i + 1]);
81
82         // TODO: Add collision constraints
83     }
84
85     // ****
86     // Simulate - Simulate verlet physics for rope
87     // ****
88     void ARopeSim::Simulate(float DeltaTime)
89     {
90         for (int i = 0; i < NumVerts; i++)
91             SimulateVert(Verts[i], DeltaTime);
92     }
93
94     // ****
95     // Render - Renders the rope
96     // ****
97     void ARopeSim::Render()
98     {
99         // TODO: Attach meshes to verts instead of debug lines.
100
101         // For each vert, draw a debug line, alternating between the colours red and blue.
102         for (int i = 0; i < NumVerts - 1; i++)
103         {
104             DrawDebugLine(GetWorld(), Verts[i].Pos, Verts[i + 1].Pos, (i&1) ? FColor::Red : FColor::Blue, false, -1.0f, 0, 5.0f);
105         }
106     }
107
108     // ****
109     // DebugDraw - Draws a shpere at the position of vertex index 'i'
110     // ****
111     void ARopeSim::DebugDraw(int i)
112     {
113         DrawDebugSphere(GetWorld(), Verts[i].Pos, 10.0f, 8, FColor::White);
114     }

```

Figure 225 – RopeSim (Part 2)

The next three screenshots show the cloth class being made by creating, deriving and naming the new class.

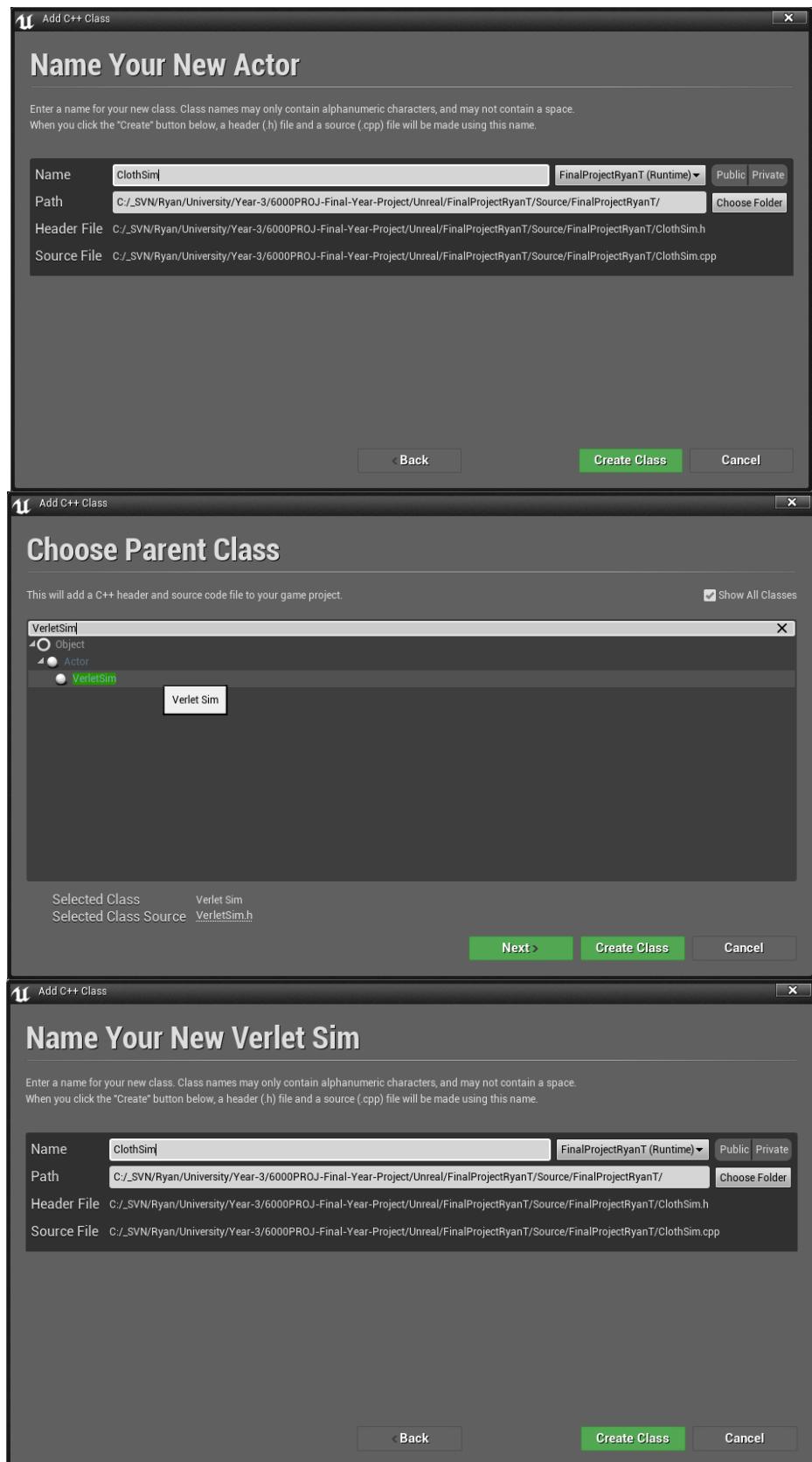


Figure 226 – Making New Class

The following screenshot shows the initial code for cloth class in ClothSim.h

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "VerletSim.h"
6 #include "ClothSim.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API AClothSim : public AVerletSim
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     AClothSim();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick(float DeltaSeconds) override;
22 };
```

Figure 227

The following screenshot shows the initial code for cloth class in ClothSim.cpp

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "ClothSim.h"
5 #include "DrawDebugHelpers.h"
6 #include "Kismet/GameplayStatics.h"
7 #include "SceneManagement.h"
8
9 // ****
10 // Constructor
11 // ****
12 AClothSim::AClothSim()
13 {
14     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
15     PrimaryActorTick.bCanEverTick = true;
16 }
17
18 // ****
19 // BeginPlay - Called when the game starts or when spawned
20 // ****
21 void AClothSim::BeginPlay()
22 {
23     Super::BeginPlay();
24 }
25
26
27 // ****
28 // Tick - Called every frame
29 // ****
30 void AClothSim::Tick(float DeltaTime)
31 {
32     Super::Tick(DeltaTime);
33 }
34
```

Figure 228

The next screenshot shows functions being declared in the ClothSim class. They include Constrain, Simulate, Render and DebugDraw. Also defined are two constants, NumVertsW and NumVertsH that define the dimensions of the cloth. Finally the actual array that will contain the vertices themselves.

```
3     #pragma once
4
5     #include "VerletSim.h"
6     #include "ClothSim.generated.h"
7
8     UCCLASS()
9     class FINALPROJECTRYANT_API AClothSim : public AVerletSim
10    {
11        GENERATED_BODY()
12
13    public:
14        // Sets default values for this actor's properties
15        AClothSim();
16
17        // Called when the game starts or when spawned
18        virtual void BeginPlay() override;
19
20        // Called every frame
21        virtual void Tick(float DeltaSeconds) override;
22
23        // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
24        // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.
25
26        // Toggles display of debug draw lines
27        UPROPERTY(EditAnywhere, Category = "Cloth Settings", meta = (DisplayName = "Enable Debug Draw"))
28        bool bEnableDebugDraw = false;
29
30    private:
31        void Constrain();
32
33        void Simulate(float DeltaTime);
34
35        void Render();
36
37        static const int NumVertsW = 40;
38
39        static const int NumVertsH = 20;
40
41        Verlet Verts[NumVertsW][NumVertsH];
42
43        void DebugDraw(int y, int z);
44    };
```

Figure 229

This screenshot shows the class constructor and BeginPlay. In BeginPlay the vertex array is initialised to the default positions.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "ClothSim.h"
5 #include "DrawDebugHelpers.h"
6 #include "Kismet/GameplayStatics.h"
7 #include "SceneManagement.h"
8
9 // ****
10 // Constructor
11 // ****
12 AClothSim::AClothSim()
13 {
14     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
15     PrimaryActorTick.bCanEverTick = true;
16 }
17
18 // ****
19 // BeginPlay - Called when the game starts or when spawned
20 // ****
21 void AClothSim::BeginPlay()
22 {
23     Super::BeginPlay();
24
25     FTransform Transform = GetActorTransform();
26     FVector SideAxis = Transform.GetUnitAxis(EAxis::Y);
27     FVector UpAxis = Transform.GetUnitAxis(EAxis::Z);
28
29     // Initialising all verts in cloth
30     FVector Pos = Transform.GetLocation();
31     for (int z = 0; z < NumVertsH; z++)
32     {
33         FVector OldPos = Pos;                      // Remember position of left side of cloth
34
35         for (int y = 0; y < NumVertsW; y++)
36         {
37             Verts[y][z].Flags = 0;
38             Verts[y][z].Pos = Pos;
39             Verts[y][z].PreviousPos = Pos;
40
41             Pos += SideAxis * ConstraintDist;        // Move across the SideAxis by ConstrainDist world units
42         }
43
44         Pos = OldPos;                           // Resets position to left side
45         Pos -= UpAxis * ConstraintDist;         // Move down the UpAxis by ConstrainDist world units
46     }
47 }
```

Figure 230

The next screenshot shows the Tick function. This is called every frame and locks the top corner vertices so they do not move. It uses the actors side and up axis from the transform to calculate the offsets for the corner positions in world space. It also calls the Simulate, Constrain and Render functions.

```
49 // ****
50 // Tick - Called every frame
51 // ****
52 void AClothSim::Tick(float DeltaTime)
53 {
54     Super::Tick(DeltaTime);
55
56     FTransform Transform = GetActorTransform();
57     FVector SideAxis = Transform.GetUnitAxis(EAxis::Y);
58     FVector UpAxis = Transform.GetUnitAxis(EAxis::Z);
59
60     // Locks top left vert of the cloth to the actor transform location and disables simulation
61     Verts[0][0].Pos = GetActorLocation();
62     Verts[0][0].Flags = 1;
63
64     // Locks top right vert of the cloth to the actor location along the Y axis and disables simulation
65     Verts[NumVertsW - 1][0].Pos = GetActorLocation() + SideAxis * ConstraintDist * (NumVertsW - 1);
66     Verts[NumVertsW - 1][0].Flags = 1;
67
68     Simulate(DeltaTime);
69     Constrain();
70     Render();
71 }
```

Figure 231

This screenshot shows the constrain function. It resolves any collisions for each vertex before constraining each vertex by a fixed distance the vertices up, down, left and right of itself.

```
74 // Constrain - Maintain equal distance between all verts
75 // ****
76 void AClothSim::Constrain()
77 {
78     for (int z = 0; z < NumVertsH; z++)
79     {
80         for (int y = 0; y < NumVertsW; y++)
81         {
82             // Resolve collision for current vert
83             if (ResolveVertCollision(Verts[y][z]))
84             {
85                 // Listener for cloth debug draw tickbox
86                 if (bEnableDebugDraw)
87                     DebugDraw(y,z);
88             }
89         }
90     }
91
92     // down
93     for (int i = 0; i < 15; i++)
94     {
95         for (int z = 0; z < NumVertsH - 1; z++)
96         {
97             for (int y = 0; y < NumVertsW; y++)
98             {
99                 Verlet &Current = Verts[y][z];
100
101                // Constrain vert below the current
102                ConstrainVert(Current, Verts[y][z + 1]);
103            }
104        }
105        // up
106        for (int i = 0; i < 15; i++)
107        {
108            for (int z = NumVertsH - 1; z >= 1; z--)
109            {
110                for (int y = 0; y < NumVertsW; y++)
111                {
112                    Verlet &Current = Verts[y][z];
113
114                    // Constrain vert above the current
115                    ConstrainVert(Current, Verts[y][z - 1]);
116                }
117            }
118            // right
119            for (int i = 0; i < 15; i++)
120            {
121                for (int z = 0; z < NumVertsH; z++)
122                {
123                    for (int y = 0; y < NumVertsW - 1; y++)
124                    {
125                        Verlet &Current = Verts[y][z];
126
127                        // Constrain vert to the right of current
128                        ConstrainVert(Current, Verts[y + 1][z]);
129                    }
130                }
131                // left
132                for (int i = 0; i < 15; i++)
133                {
134                    for (int z = 0; z < NumVertsH; z++)
135                    {
136                        for (int y = NumVertsW - 1; y >= 1; y--)
137                        {
138                            Verlet &Current = Verts[y][z];
139
140                            // Constrain vert to the left of current
141                            ConstrainVert(Current, Verts[y - 1][z]);
142                        }
143
144                    }
145
146                }
147
148            }
149
150        }
151    }
152
153    // TODO: Add collision constraints
154 }
```

Figure 232

This screenshot shows the Simulate, Render and DebugDraw functions. Simulate simply calls the base class's SimulateVert function for each vertex in the array. The Render function draws the whole mesh using debug lines. This will be replaced by a real mesh later on. DebugDraw is a helper function used to draw a sphere at a particular vertex position specified by the x & y arguments.

```

143
144 // ****
145 // Simulate - Simulate verlet physics for cloth
146 // ****
147 void AClothSim::Simulate(float DeltaTime)
148 {
149     for (int z = 0; z < NumVertsH; z++)
150         for (int y = 0; y < NumVertsW; y++)
151             SimulateVert(Verts[y][z], DeltaTime);
152 }
153
154 // ****
155 // Render - Renders the cloth
156 // ****
157 void AClothSim::Render()
158 {
159     // TODO: Attach meshes to verts instead of debug lines.
160
161     // For each vert, draw a debug line, alternating between the colours red and blue.
162     for (int z = 0; z < NumVertsH - 1; z++)
163     {
164         for (int y = 0; y < NumVertsW - 1; y++)
165         {
166             float Length = (Verts[y][z].Pos - Verts[y + 1][z].Pos).Size();
167
168             FColor Colour = FColor::Green;
169
170             if (fabs(Length - ConstraintDist) > 0.1f)
171                 Colour = FColor::Black;
172
173             // Draw top line
174             DrawDebugLine(GetWorld(), Verts[y][z].Pos, Verts[y + 1][z].Pos, Colour, false, -1.0f, 0, 2.0f);
175
176             // Draw left line
177             DrawDebugLine(GetWorld(), Verts[y][z].Pos, Verts[y][z + 1].Pos, FColor::Red, false, -1.0f, 0, 2.0f);
178         }
179     }
180 }
181
182 // ****
183 // DebugDraw - Draws debug spheres
184 // ****
185 void AClothSim::DebugDraw(int y, int z)
186 {
187     DrawDebugSphere(GetWorld(), Verts[y][z].Pos, 10.0f, 8, FColor::White);
188 }
```

Figure 233

The screenshot shows gameplay of cloth. The Simulation is not working correctly as the cloth is sagging far too much.

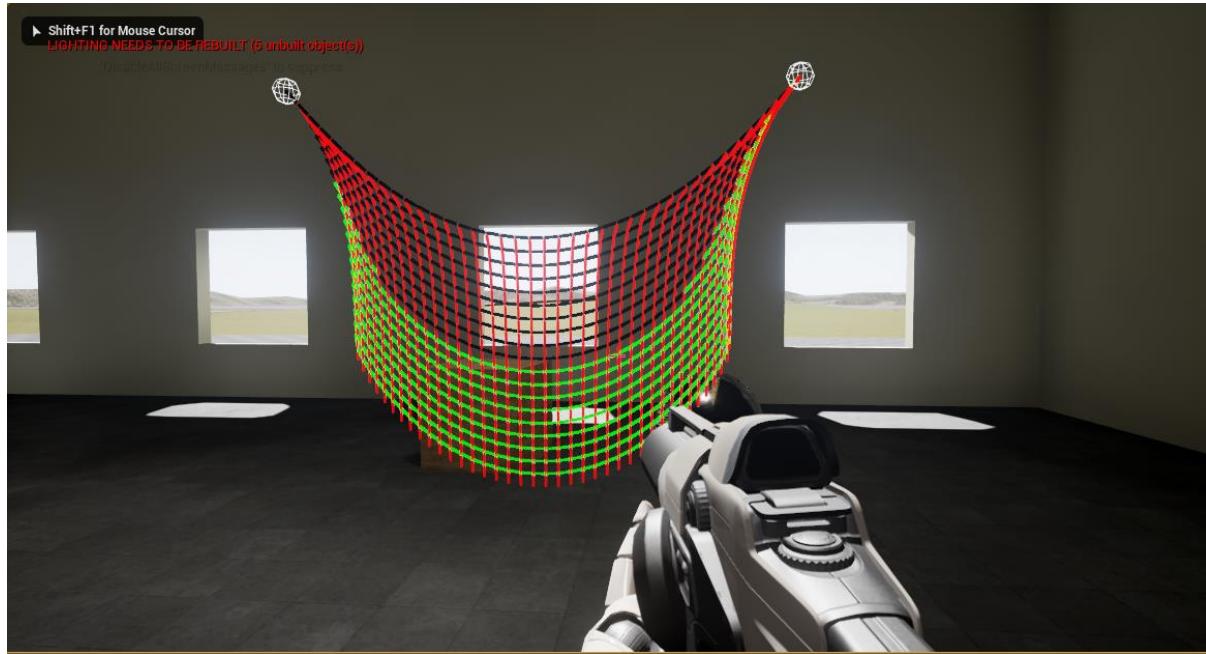


Figure 234

This screenshot shows changes to the constrain function to fix the sagging problem.

```
110
111 // ****
112 // Constrain - Maintain equal distance between all verts
113 // ****
114 void AClothSim::Constrain()
115 {
116     for (int z = 0; z < NumVertsH; z++)
117     {
118         for (int y = 0; y < NumVertsW; y++)
119         {
120             // Resolve collision for current vert
121             if (ResolveVertCollision(Verts[y][z]))
122             {
123                 // Listener for rope debug draw tickbox
124                 if (bEnableDebugDraw)
125                     DebugDraw(y, z);
126             }
127         }
128     }
129     for (int i = 0; i < ConstraintList.Num() ; i++)
130     {
131         ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
132     }
133     // TODO: Add collision constraints
134 }
```

Figure 235

The next screenshot show Gameplay of the issue fixed.



Figure 236

This screenshot shows changes to the rope. Modified the rope tick function so it does multiple iterations of the constrain function. This is to improve the stability of the simulation and stop jittering.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "RopeSim.h"
5 #include "DrawDebugHelpers.h"
6 #include "Kismet/GameplayStatics.h"
7 #include "SceneManagement.h"
8
9 #define EDGE_LENGTH 10.0
10
11 // ****
12 // Constructor
13 // ****
14 ARopeSim::ARopeSim()
15 {
16     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
17     PrimaryActorTick.bCanEverTick = true;
18 }
19
20 // ****
21 // BeginPlay - Called when the game starts or when spawned
22 // ****
23 void ARopeSim::BeginPlay()
24 {
25     Super::BeginPlay();
26
27     // Initialising all verts in rope
28     FVector Pos = GetActorLocation();
29
30     for (int i = 0; i < NumVerts; i++)
31     {
32         Verts[i].Flags = 0;
33         Verts[i].Pos = Pos;
34         Verts[i].PreviousPos = Pos;
35         Pos.Z -= EDGE_LENGTH;           // Move down the rope to the next vert
36
37         CreateCollisionComponent(Verts[i]);
38     }
39 }
40
41 // ****
42 // Tick - Called every frame
43 // ****
44 void ARopeSim::Tick(float DeltaTime)
45 {
46     Super::Tick(DeltaTime);
47
48     // Locks first vert of the rope to the actor transform location and disables simulation
49     Verts[0].Pos = GetActorLocation();
50     Verts[0].Flags = 1;
51
52     // Simulate
53     Simulate(DeltaTime);
54
55     // Constrain
56     for (int i = 0; i < 15; i++)
57         Constrain();
58
59     for (int i = 0; i < NumVerts; i++)
60     {
61         if (Verts[i].SphereComponent)
62             Verts[i].SphereComponent->SetWorldLocation(Verts[i].Pos);
63     }
64
65     // Render
66     Render();
67 }
68
```

Figure 237

Changes made to the rope constrain function. It now constrains forward through the vertex list and then backwards to further improve stability.

```
68
69     // ****
70     // Constrain - Maintain equal distance between all verts
71     // ****
72     void ARopeSim::Constrain()
73     {
74         for (int i = 0; i < NumVerts; i++)
75         {
76             // Resolve collision for current vert
77             if (ResolveVertCollision(Verts[i]))
78             {
79                 // Listener for rope debug draw tickbox
80                 if (bEnableDebugDraw)
81                     DebugDraw(i);
82             }
83         }
84
85         for (int i = 0; i < NumVerts - 1; i++)
86             ConstrainVert(Verts[i], Verts[i + 1], EDGE_LENGTH);
87
88         for (int i = NumVerts - 1; i >= 1; i--)
89             ConstrainVert(Verts[i], Verts[i - 1], EDGE_LENGTH);
90
91         // TODO: Add collision constraints
92     }
93
94     // ****
95     // Simulate - Simulate verlet physics for rope
96     // ****
97     void ARopeSim::Simulate(float DeltaTime)
98     {
99         for (int i = 0; i < NumVerts; i++)
100            SimulateVert(Verts[i], DeltaTime);
101    }
102
103    // ****
104    // Render - Renders the rope
105    // ****
106    void ARopeSim::Render()
107    {
108        // TODO: Attach meshes to verts instead of debug lines.
109
110        // For each vert, draw a debug line, alternating between the colours red and blue.
111        for (int i = 0; i < NumVerts - 1; i++)
112        {
113            float Length = (Verts[i].Pos - Verts[i + 1].Pos).Size();
114
115            FColor Colour = (i & 1) ? FColor::Red : FColor::Blue;
116
117            DrawDebugLine(GetWorld(), Verts[i].Pos, Verts[i + 1].Pos, Colour, false, -1.0f, 0, 5.0f);
118        }
119    }
120
121    void ARopeSim::DebugDraw(int i)
122    {
123        DrawDebugSphere(GetWorld(), Verts[i].Pos, 10.0f, 8, FColor::White);
124    }
```

Figure 238

The next three screenshots show the addition of a constraint list. Lines that connect 2 vertices together and handle their constraint. They constrain horizontally, vertically as well as diagonally.

```

19 // ****
20 // BeginPlay - Called when the game starts or when spawned
21 // ****
22 void AClothSim::BeginPlay()
23 {
24     Super::BeginPlay();
25
26     FTransform Transform = GetActorTransform();
27     FVector SideAxis = Transform.GetUnitAxis(EAxis::Y);
28     FVector UpAxis = Transform.GetUnitAxis(EAxis::Z);
29
30     // Initialising all verts in rope
31     FVector Pos = Transform.GetLocation();
32     for (int z = 0; z < NumVertsH; z++)
33     {
34         FVector OldPos;                                // Remember position of left side of cloth
35
36         for (int y = 0; y < NumVertsW; y++)
37         {
38             Verts[y][z].Flags = 0;
39             Verts[y][z].Pos = Pos;
40             Verts[y][z].PreviousPos = Pos;
41
42             Pos += SideAxis * EDGE_LENGTH;           // Move across the SideAxis by ConstrainDist world units
43
44             CreateCollisionComponent(Verts[y][z]);
45         }
46         Pos = OldPos;                                // Resets position to left side
47         Pos -= UpAxis * EDGE_LENGTH;                // Move down the UpAxis by ConstrainDist world units
48     }
49
50
51     // Connecting immediate neighbor particles with constraints (distance 1 and sqrt(2) in the grid)
52     for (int x = 0; x<NumVertsW; x++)
53     {
54         for (int y = 0; y<NumVertsH; y++)
55         {
56             if (x<NumVertsW - 1)          CreateConstraint(&Verts[x]      [y] , &Verts[x + 1] [y]      );
57             if (y<NumVertsH - 1)          CreateConstraint(&Verts[x]      [y] , &Verts[x]      [y + 1] );
58             if (x<NumVertsW - 1 && y<NumVertsH - 1) CreateConstraint(&Verts[x]      [y] , &Verts[x + 1] [y + 1] );
59             if (x<NumVertsW - 1 && y<NumVertsH - 1) CreateConstraint(&Verts[x + 1] [y] , &Verts[x]      [y + 1] );
60         }
61     }
62
63
64     // Connecting secondary neighbors with constraints (distance 2 and sqrt(4) in the grid)
65     for (int x = 0; x<NumVertsW; x++)
66     {
67         for (int y = 0; y<NumVertsH; y++)
68         {
69             if (x<NumVertsW - 2)          CreateConstraint(&Verts[x]      [y] , &Verts[x + 2] [y]      );
70             if (y<NumVertsH - 2)          CreateConstraint(&Verts[x]      [y] , &Verts[x]      [y + 2] );
71             if (x<NumVertsW - 2 && y<NumVertsH - 2) CreateConstraint(&Verts[x]      [y] , &Verts[x + 2] [y + 2] );
72             if (x<NumVertsW - 2 && y<NumVertsH - 2) CreateConstraint(&Verts[x + 2] [y] , &Verts[x]      [y + 2] );
73         }
74     }
75
76 }
77

```

```

78 // ****
79 // Tick - Called every frame
80 // ****
81 void AClothSim::Tick(float DeltaTime)
82 {
83     //DeltaTime /= 10.0f;
84
85     Super::Tick(DeltaTime);
86
87     FTransform Transform = GetActorTransform();
88     FVector SideAxis = Transform.GetUnitAxis(EAxis::Y);
89     FVector UpAxis = Transform.GetUnitAxis(EAxis::Z);
90
91     // Locks top left vert of the cloth to the actor transform location and disables simulation
92     Verts[0][0].Pos = GetActorLocation();
93     Verts[0][0].Flags = 1;
94
95     // Locks top right vert of the cloth to the actor location along the Y axis and disables simulation
96     Verts[NumVertsW - 1][0].Pos = GetActorLocation() + SideAxis * EDGE_LENGTH * NumVertsW;
97     Verts[NumVertsW - 1][0].Flags = 1;
98
99     Simulate(DeltaTime);
100
101    for (int i = 0; i < 15; i++)
102        Constrain();
103
104    for (int z = 0; z < NumVertsH; z++)
105    {
106        for (int y = 0; y < NumVertsW; y++)
107        {
108            if (Verts[y][z].SphereComponent)
109                Verts[y][z].SphereComponent->SetWorldLocation(Verts[y][z].Pos);
110        }
111    }
112
113    Render();
114 }
115
116 // ****
117 // Constrain - Maintain equal distance between all verts
118 // ****
119 void AClothSim::Constrain()
120 {
121     for (int z = 0; z < NumVertsH; z++)
122     {
123         for (int y = 0; y < NumVertsW; y++)
124         {
125             // Resolve collision for current vert
126             if (ResolveVertCollision(Verts[y][z]))
127             {
128                 // Listener for rope debug draw tickbox
129                 if (bEnableDebugDraw)
130                     DebugDraw(y, z);
131             }
132         }
133     }
134
135     for (int i = 0; i < ConstraintList.Num(); i++)
136     {
137         ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
138     }
139
140     // TODO: Add collision constraints
141 }

```

```

142
143     // ****
144     // Simulate - Simulate verlet physics for rope
145     // ****
146     void AClothSim::Simulate(float DeltaTime)
147     {
148         for (int z = 0; z < NumVertsH; z++)
149             for (int y = 0; y < NumVertsW; y++)
150                 SimulateVert(Verts[y][z], DeltaTime);
151     }
152
153     // ****
154     // Render - Renders the rope
155     // ****
156     void AClothSim::Render()
157     {
158
159         // TODO: Attach meshes to verts instead of debug lines.
160
161         // For each vert, draw a debug line, alternating between the colours red and blue.
162         for (int z = 0; z < NumVertsH - 1; z++)
163         {
164             for (int y = 0; y < NumVertsW - 1; y++)
165             {
166                 float Length = (Verts[y][z].Pos - Verts[y + 1][z].Pos).Size();
167
168                 FColor Colour = FColor::Green;
169
170                 if (fabs(Length - EDGE_LENGTH) > 0.15f)
171                     Colour = FColor::Black;
172
173                 // Draw top line
174                 DrawDebugLine(GetWorld(), Verts[y][z].Pos, Verts[y + 1][z].Pos, Colour, false, -1.0f, 0, 2.0f);
175
176                 Length = (Verts[y][z].Pos - Verts[y][z + 1].Pos).Size();
177                 Colour = FColor::Red;
178
179                 if (fabs(Length - EDGE_LENGTH) > 0.15f)
180                     Colour = FColor::Black;
181
182                 // Draw left line
183                 DrawDebugLine(GetWorld(), Verts[y][z].Pos, Verts[y][z + 1].Pos, Colour, false, -1.0f, 0, 2.0f);
184             }
185         }
186     }
187
188     void AClothSim::DebugDraw(int y, int z)
189     {
190         DrawDebugSphere(GetWorld(), Verts[y][z].Pos, 10.0f, 8, FColor::White);
191     }

```

Figure 239 – ClothSim (Three Images)

The next 2 screenshots show the ropes and cloth blowing in the wind with new constraint code applied.

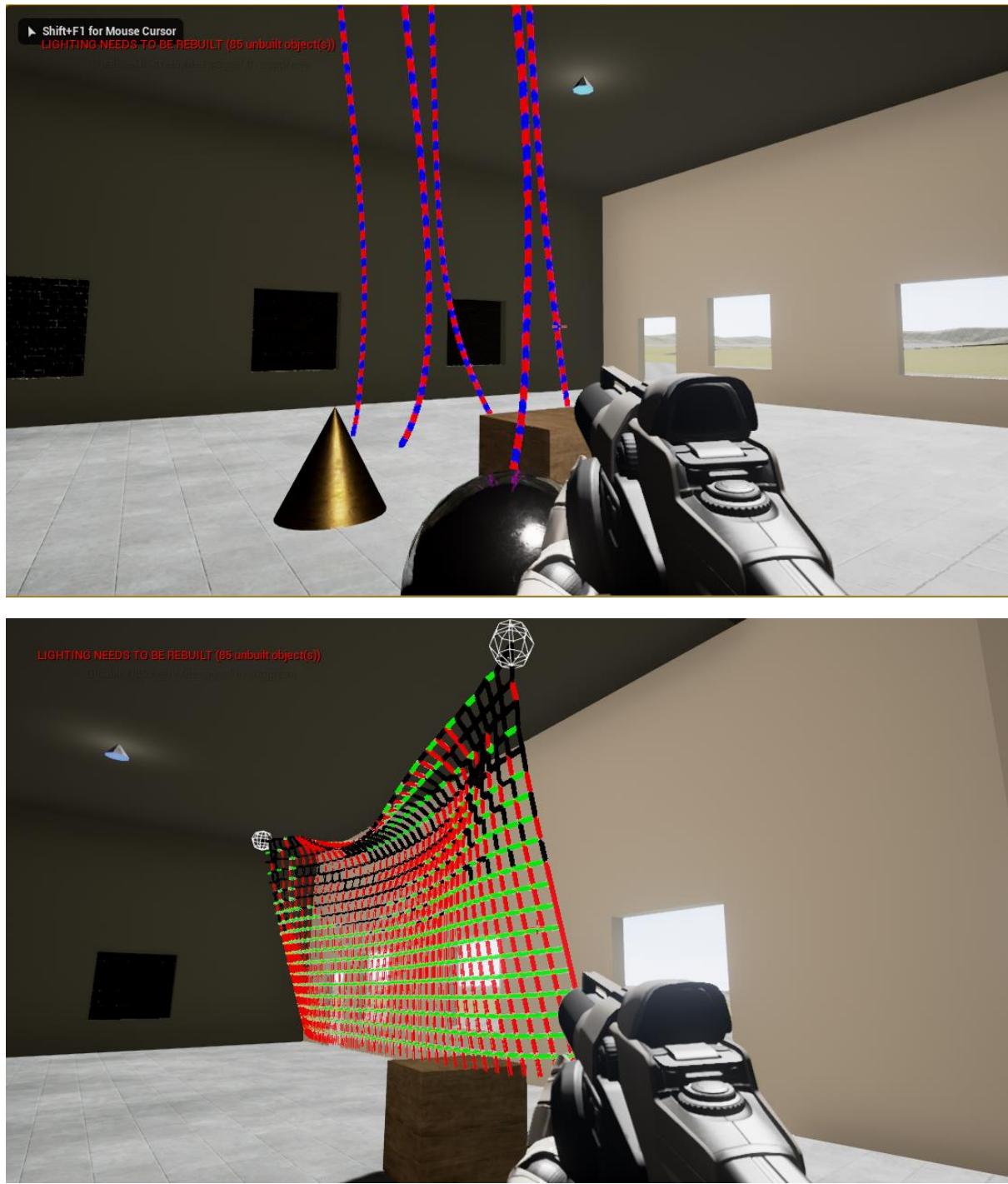


Figure 240 – Rope & Cloth

This shows the cloth blowing in the wind in wireframe.

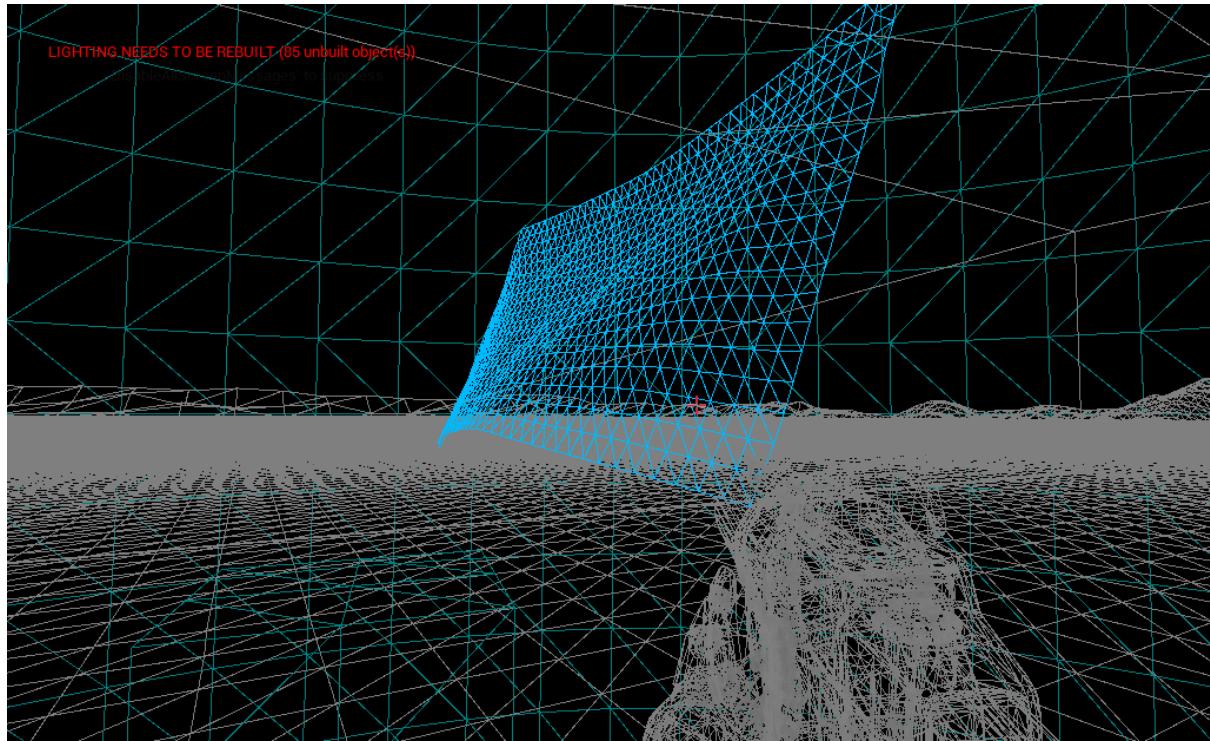


Figure 241

This function is the start of the cloth collision code. It calls ResolveVertCollision in the base class for each vertex in the cloth.

```
102
103     // ****
104     // ResolveCollision - Resolve collisions for all verts (Based on the velocity)
105     // ****
106 void AClothSim::ResolveCollision()
107 {
108     if ( !bEnableCollision )
109         return;
110
111     SCOPE_CYCLE_COUNTER(STAT_ClothResolveCollision);
112
113     for (int z = 0; z < NumVertsH; z++)
114     {
115         for (int y = 0; y < NumVertsW; y++)
116         {
117             // Resolve collision for current vert
118             if (ResolveVertCollision(Verts[y][z]))
119             {
120                 // Listener for rope debug draw tickbox
121                 if (bEnableDebugDraw)
122                     DebugDraw(y, z);
123             }
124         }
125     }
126 }
```

Figure 242

And the same for the rope class.

```
104
105     // ****
106     // ResolveCollision - Resolve collisions for all verts (Based on the velocity)
107     // ****
108 void ARopeSim::ResolveCollision()
109 {
110     if ( !bEnableCollision )
111         return;
112
113     SCOPE_CYCLE_COUNTER(STAT_RopeResolveCollision);
114
115     for (int i = 0; i < NumVerts; i++)
116     {
117         // Resolve collision for current vert
118         if (ResolveVertCollision(Verts[i]))
119         {
120             // Listener for rope debug draw tickbox
121             if (bEnableDebugDraw)
122                 DebugDraw(i);
123         }
124     }
125 }
```

Figure 243

Here is the ResolveVertCollision in the base class.

```
184
185     // ****
186     // ResolveVertCollision - Handles collision for verts
187     // ****
188 bool AVerletSim::ResolveVertCollision(Verlet& Vert)
189 {
190     FHitResult HitResult;
191     FCollisionQueryParams Params;
192     FCollisionObjectQueryParams ObjParams;
193
194     ObjParams.AddObjectTypeToQuery( ECC_WorldStatic );
195     ObjParams.AddObjectTypeToQuery( ECC_WorldDynamic );
196
197     if (GetWorld()->LineTraceSingleByObjectType( HitResult, Vert.PreviousPos, Vert.Pos, ObjParams, Params))
198     {
199         FVector Impulse = -HitResult.ImpactNormal * 40.0f;
200
201         FVector NewPos = HitResult.ImpactPoint + HitResult.ImpactNormal * 4.0f;
202
203         FVector Force = HitResult.ImpactNormal * 10.0f;
204
205         Vert.Pos      = NewPos;
206         Vert.PreviousPos = NewPos;
207         Vert.AddForce( Force );
208
209         DrawDebugLine(GetWorld(), HitResult.ImpactPoint, HitResult.ImpactPoint + Force, FColor::White );
210
211         if ( HitResult.GetComponent() )
212         {
213             // If it's movable, add an impulse to it
214             if ( HitResult.GetComponent()->IsSimulatingPhysics() && HitResult.GetComponent()->Mobility == EComponentMobility::Movable )
215                 HitResult.GetComponent()->AddImpulseAtLocation( Impulse, HitResult.ImpactPoint );
216         }
217     }
218
219     return false;
220 }
221 }
```

Figure 244

The following five screenshots show functions to build and update a procedural mesh component so that the rope and cloth has a real mesh not just wireframe.

```
204 // ****
205 void ARopeSim::BuildMesh()
206 {
207     MeshComponent->ClearAllMeshSections();      // Clear any old meshes
208
209     TArray <FVector>           Vertices;
210     TArray <int32>            Triangles;
211     TArray <FVector>           Normals;
212     TArray <FVector2D>         UV0;
213     TArray <FColor>            VertexColor;
214     TArray <FProcMeshTangent> VertexCTangent;
215
216     // Build the vertex list
217     for (int i = 0; i < NumVerts; i++)
218     {
219         float Rad = 0.0f;
220         float RadDelta = (PI*2.0f) / float(RopeSegments);    // Circumference divided by the number of segments (In Radians)
221         for ( int s = 0 ; s < RopeSegments ; s++ )
222         {
223             FVector Offset;
224             Offset.X = cosf( Rad ) * RopeThickness;
225             Offset.Y = sinf( Rad ) * RopeThickness;
226             Offset.Z = 0.0f;                                // Always on horizontal plane
227
228             FVector Pos = (Verts[i].Pos + Offset);
229
230             Vertices.Add( Pos );
231
232             FVector Normal = Offset;
233             Normal.Normalize();
234             Normals.Add( Normal );
235
236             FVector2D UV = FVector2D(s,i);
237             UV.X /= float(RopeSegments);        // Horizontal: 0.0 to 1.0 mapped around the rope
238             UV.Y /= float(NumVerts);           // Vertical: 0.0 to 1.0 from top of rope to bottom
239             UV0.Add(UV);
240
241             Rad += RadDelta;                // Next segment in radians
242         }
243     }
244
245     for (int i = 0; i < NumVerts-1; i++)
246     {
247         for ( int s = 0 ; s < RopeSegments ; s++ )
248         {
249             int BaseOffset = (i*RopeSegments) + s;
250
251             Triangles.Add( BaseOffset + 0 );
252             Triangles.Add( BaseOffset + RopeSegments );    // +RopeSegments moved down the rope to the next verts segment verts
253             Triangles.Add( BaseOffset + RopeSegments + 1 );
254
255             Triangles.Add( BaseOffset + RopeSegments+1 );
256             Triangles.Add( BaseOffset + 1 );
257             Triangles.Add( BaseOffset + 0 );
258         }
259     }
260
261     bool bCreateCollision = false;
262     MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );
263
264     MeshComponent->SetMaterial(0,MeshMaterial);
265     MeshComponent->SetWorldTransform( FTransform() );      // Set transform to ID (Zero Position & Zero Rotation) as all verts are in world space.
266 }
```

```

267     // ****
268     // UpdateMesh - Update the textured mesh
269     // ****
270
271     void ARopeSim::UpdateMesh()
272     {
273         SCOPE_CYCLE_COUNTER(STAT_RopeUpdateMesh);
274
275         TArray <FVector> Vertices;
276         TArray <FVector> Normals;
277         TArray <FVector2D> UV0;
278         TArray <FColor> VertexColor;
279         TArray <FProcMeshTangent> VertexCTangent;
280
281         FVector SurfaceNormals[NumVerts*MAX_ROPE_SEGMENTS];
282
283         for (int i = 0; i < NumVerts; i++)
284         {
285             //FVector Up      = FVector( 0,0,1 );
286             //FVector Side   = FVector( 0,1,0 );
287             //FVector Forward = FVector( 1,0,0 );
288
289             FVector Up, Side, Forward;
290
291             // Get the vector pointing up the rope at this vert
292             if (i < NumVerts-1)
293                 Up = Verts[i].Pos - Verts[i+1].Pos;
294
295             else
296                 Up = Verts[i-1].Pos - Verts[i].Pos;
297             Up.Normalize();
298
299             Side    = FVector::CrossProduct( Up, FVector(0,1,0) ); // Get the side vector (Perpendicular to the world Y Axis (Forward) )
300             Forward = FVector::CrossProduct( Up, Side );           // Get the forward vector (Perpendicular Side axis and Up axis)
301             Side    = FVector::CrossProduct( Up, Forward );        // Get the 'real' side vector (Perpendicular to Up and Forward)
302
303             float Rad = 0.0f;
304             float RadDelta = (PI*2.0f) / float(RopeSegments);       // Circumference divided by the number of segments (In Radians)
305
306             for (int s = 0; s < RopeSegments; s++)
307             {
308
309                 FVector Offset    = Side    * cosf(Rad) * RopeThickness;
310                 Offset      += Forward * sinf(Rad) * RopeThickness;
311                 FVector Pos = (Verts[i].Pos + Offset);
312
313                 Vertices.Add(Pos);
314
315                 SurfaceNormals[(i*RopeSegments)+s] = Offset;
316                 SurfaceNormals[(i*RopeSegments)+s].Normalize();
317
318                 Rad += RadDelta;           // Next segment in radians
319             }
320         }
321
322         // Build the vertex normals from the surface normals
323         for (int i = 0; i < NumVerts; i++)
324         {
325             float Rad = 0.0f;
326             float RadDelta = (PI*2.0f) / float(RopeSegments);       // Circumference divided by the number of segments (In Radians)
327
328             for (int s = 0; s < RopeSegments; s++)
329             {
330                 FVector Normal(0.0f,0.0f,0.0f);
331
332                 if ( i > 0 )
333                     Normal += SurfaceNormals[((i-1)*RopeSegments)+s];
334                 if ( i < (NumVerts-1) )
335                     Normal += SurfaceNormals[((i+1)*RopeSegments)+s];
336
337                 if ( s > 0 )
338                     Normal += SurfaceNormals[(i*RopeSegments)+(s-1)];
339                 if ( s < (RopeSegments-1) )
340                     Normal += SurfaceNormals[(i*RopeSegments)+(s+1)];
341
342                 Normal.Normalize();
343                 Normals.Add(Normal);
344             }
345
346             MeshComponent->UpdateMeshSection( 0, Vertices, Normals, UV0, VertexColor, VertexCTangent );
347         }
348     }

```

```

209     // ****
210     // BuildMesh - Build the textured mesh
211     // ****
212     void AClothSim::BuildMesh()
213     {
214         MeshComponent->ClearAllMeshSections();      // Clear any old meshes
215
216         TArray <FVector>           Vertices;
217         TArray <int32>             Triangles;
218         TArray <FVector>           Normals;
219         TArray <FVector2D>          UV0;
220         TArray <FColor>            VertexColor;
221         TArray <FPProcMeshTangent> VertexCTangent;
222
223         // Build the vertex list
224         for (int z = 0; z < NumVertsH; z++)
225         {
226             for (int y = 0; y < NumVertsW; y++)
227             {
228                 Vertices.Add( Verts[y][z].Pos );
229
230                 Normals.Add( FVector(1,0,0) );
231
232                 FVector2D UV = FVector2D(y,z);
233                 UV.X /= float(NumVertsW);
234                 UV.Y /= float(NumVertsH);
235                 UV0.Add(UV);
236             }
237         }
238
239         // Build the triangle list. Each triangle has 3 x Indices into the above vertex list.
240         //
241         //    0----1
242         //    | \   |
243         //    |  \  |
244         //    |   \ |
245         //    |    \| |
246         //    W+0--W+1
247         //
248         for (int z = 0; z < NumVertsH-1; z++)
249         {
250             for (int y = 0; y < NumVertsW-1; y++)
251             {
252                 int32 BaseIndex = y+(z*NumVertsW);
253
254                 Triangles.Add( BaseIndex + 0 );
255                 Triangles.Add( BaseIndex + NumVertsW );
256                 Triangles.Add( BaseIndex + NumVertsW + 1 );
257
258                 Triangles.Add( BaseIndex + NumVertsW+1 );
259                 Triangles.Add( BaseIndex + 1 );
260                 Triangles.Add( BaseIndex + 0 );
261             }
262         }
263
264         bool bCreateCollision = false;
265         MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );
266
267         MeshComponent->SetMaterial(0,MeshMaterial);
268     }

```

```

272 // ****
273 void AClothSim::UpdateMesh()
274 {
275     SCOPE_CYCLE_COUNTER(STAT_ClothUpdateMesh);
276
277     TArray < FVector > Vertices;
278     TArray < FVector > Normals;
279     TArray < FVector2D > UV0;
280     TArray < FColor > VertexColor;
281     TArray < FProcMeshTangent > VertexCTangent;
282
283     FVector SurfaceNormals[NumVertsW][NumVertsH];
284
285     for (int z = 0; z < NumVertsH; z++)
286     {
287         for (int y = 0; y < NumVertsW; y++)
288         {
289             Vertices.Add(Verts[y][z].Pos);
290
291             FVector Normal, V1, V2;
292
293             if (y < (NumVertsW - 1))
294                 V1 = Verts[y + 1][z].Pos - Verts[y][z].Pos;
295
296             else
297                 V1 = Verts[y][z].Pos - Verts[y - 1][z].Pos;
298
299             if (z < (NumVertsH - 1))
300                 V2 = Verts[y][z + 1].Pos - Verts[y][z].Pos;
301
302             else
303                 V2 = Verts[y][z].Pos - Verts[y][z - 1].Pos;
304
305             SurfaceNormals[y][z] = FVector::CrossProduct(V1, V2);
306             SurfaceNormals[y][z].Normalize();
307
308         }
309     }
310
311     // Build the vertex normals from the surface normals
312     for (int z = 0; z < NumVertsH; z++)
313     {
314         for (int y = 0; y < NumVertsW; y++)
315         {
316             FVector Normal(0.0f, 0.0f, 0.0f);
317
318             if (z > 0)
319                 Normal += SurfaceNormals[y][z - 1];
320             if (z < (NumVertsH - 1))
321                 Normal += SurfaceNormals[y][z + 1];
322
323             if (y > 0)
324                 Normal += SurfaceNormals[y - 1][z];
325             if (y < (NumVertsW - 1))
326                 Normal += SurfaceNormals[y + 1][z];
327
328             Normal.Normalize();
329             Normals.Add(Normal);
330         }
331     }
332
333     MeshComponent->UpdateMeshSection(0, Vertices, Normals, UV0, VertexColor, VertexCTangent);
334 }

```

Figure 245 – Build Mesh (5 Images)

Adding code to VerletSim header file in preparation for adding ability to dynamically attach and lock points on ropes and cloth.

```
10 // ****
11 // FVerletAttachment - For adding dynamic attach/lock points to rope/cloth
12 // ****
13 USTRUCT()
14 struct FVerletAttachment
15 {
16     GENERATED_BODY()
17
18     UPROPERTY(EditAnywhere, Category = "Verlet Settings")
19         bool             bRelativeToActor    = true;
20
21     UPROPERTY(EditAnywhere, Category = "Verlet Settings", Meta = (MakeEditWidget = true))
22         FVector          AttachPos        = FVector(0.0f, 0.0f, 0.0f);
23
24     UPROPERTY(EditAnywhere, Category = "Verlet Settings")
25         int32            VertIndex       = 0;
26
27 };
28
```

Figure 246

Code to dynamically attach and lock points on ropes.

```
72 // ****
73 // UpdateLockedVerts - Update any verts that are locked
74 // ****
75 void ARopeSim::UpdateLockedVerts()
76 {
77     FTransform Transform = GetActorTransform();
78
79     for (int i = 0; i < AttachmentList.Num(); i++)
80     {
81         int Index = AttachmentList[i].VertIndex;
82
83         if (Index >= 0 && Index < NumVerts)
84         {
85             FVector Pos = AttachmentList[i].AttachPos;
86
87             if (AttachmentList[i].bRelativeToActor)
88                 Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List] positions are in local space
89
90             Verts[Index].Pos = Pos;
91             Verts[Index].PreviousPos = Pos;
92             Verts[Index].Flags = 1;
93         }
94     }
95 }
96
```

Figure 247

Code to dynamically attach and lock points on cloth.

```
386
387     // ****
388     // UpdateLockedVerts - Update any verts that are locked
389     // ****
390     void AClothSim::UpdateLockedVerts()
391     {
392         FTransform Transform = GetActorTransform();
393
394         for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
395         {
396             int Index = AttachmentList[i].VertIndex;
397
398             if ( Index >= 0 && Index < NumVertsW*NumVertsH )
399             {
400                 int Horiz = Index % NumVertsW;
401                 int Verti = Index - Horiz;
402
403                 FVector Pos = AttachmentList[i].AttachPos;
404                 if ( AttachmentList[i].bRelativeToActor )
405                     Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List] positions are in local space
406
407                 Verts[Horiz][Verti].Pos = Pos;
408                 Verts[Horiz][Verti].PreviousPos = Pos;
409                 Verts[Horiz][Verti].Flags = 1;
410             }
411         }
412     }
```

Figure 248

The next four screenshots show adding code to VerletSim header file in preparation for fixing an issue where the cloth and rope don't rotate properly. This involves setting the objects transform to world position 0, 0, 0 then handling all vertex positions in world space.

```
110
111     // ****
112     // BeginPlay - Called when the game starts or when spawned
113     // ****
114     void AVerletSim::BeginPlay()
115     {
116         Super::BeginPlay();
117
118         // Finds the first wind source object in the level
119         if (!Wind)
120         {
121             // TODO: COMMENT
122             TArray<AActor*> WindList;
123             UGameplayStatics::GetAllActorsOfClass(GetWorld(), AWindSim::StaticClass(), WindList);
124
125             if (WindList.Num())
126                 Wind = Cast<AWindSim>(WindList[0]);
127         }
128
129         MeshComponent = NewObject<UProceduralMeshComponent>(this, NAME_None);
130         MeshComponent->RegisterComponent();
131         MeshComponent->SetWorldTransform(FTransform()); // Set transform to Position(0,0,0) and Rotation(0,0,0)
132
133         // KeepWorldTransform means that the component is not relative to the AVerletSim actor, it has its own worldspace position
134         MeshComponent->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);
135         LastAddedComponent = MeshComponent;
136         MeshComponent->OnComponentHit.AddDynamic(this, &AVerletSim::OnHitCallback);
137     }
138
139     // ****
140     // Tick - Called every frame
141     // ****
142     void AVerletSim::Tick(float DeltaTime)
143     {
144         Super::Tick(DeltaTime);
145
146         MeshComponent->SetWorldTransform(FTransform()); // Set transform to Position(0,0,0) and Rotation(0,0,0)
147     }
```

```
72
73     // ****
74     // UpdateLockedVerts - Update any verts that are locked
75     // ****
76     void ARopeSim::UpdateLockedVerts()
77     {
78         FTransform Transform = GetActorTransform();
79
80         for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
81         {
82             int Index = AttachmentList[i].VertIndex;
83
84             if ( Index >= 0 && Index < NumVerts )
85             {
86                 FVector Pos = AttachmentList[i].AttachPos;
87
88                 if ( AttachmentList[i].bRelativeToActor )
89                     Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List positions are in local space
90
91                 Verts[Index].Pos = Pos;
92                 Verts[Index].PreviousPos = Pos;
93                 Verts[Index].Flags = 1;
94             }
95         }
96     }
```

```
366
367     // ****
368     // BuildConstraints - Build constraints for all verts
369     // ****
370     void ARopeSim::BuildConstraints()
371     {
372         for (int i = 0; i < NumVerts-1; i++)
373             CreateConstraint(&Verts[i], &Verts[i+1] );
374     }
375
376
377
```

```
382
383     // ****
384     // UpdateLockedVerts - Update any verts that are locked
385     // ****
386     void AClothSim::UpdateLockedVerts()
387     {
388         FTransform Transform = GetActorTransform();
389
390         for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
391         {
392             int Index = AttachmentList[i].VertIndex;
393
394             if ( Index >= 0 && Index < NumVertsW*NumVertsH )
395             {
396                 int Horiz = Index % NumVertsW;
397                 int Verti = Index - Horiz;
398
399                 FVector Pos = AttachmentList[i].AttachPos;
400                 if ( AttachmentList[i].bRelativeToActor )
401                     Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List positions are in local space
402
403                 Verts[Horiz][Verti].Pos = Pos;
404                 Verts[Horiz][Verti].PreviousPos = Pos;
405                 Verts[Horiz][Verti].Flags = 1;
406             }
407         }
408     }
```

Figure 249 – Fixing Rope (4 Images)

The next three screenshots show adding code to VerletSim class to enable triggering of an Unreal Engine Level Sequences for ropes and cloth when player enters an attached trigger volume.

The header file:

```

201 UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
202 TArray<FVerletAttachment> AttachmentList;
203
204 UPROPERTY()
205 UTextContainerComponent* TextContainer = nullptr;
206
207 UPROPERTY(EditAnywhere, Category = "Verlet Settings|Trigger")
208 ULevelSequence* LevelSequence = nullptr;
209
210 UPROPERTY()
211 ULevelSequencePlayer* SequencePlayer = nullptr;
212
213 UPROPERTY(EditAnywhere, Category = "Verlet Settings|Trigger")
214 bool bEnableTriggerVolume = false;
215
216 UPROPERTY(EditAnywhere)
217 UBoxComponent* TriggerVolume = nullptr;
218
219
220 protected:
221
222 void SimulateVert(Verlet& Vert, float DeltaTime);
223 bool ResolveVertCollision(Verlet& Vert);
224 void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
225
226 void OnEndSequenceEvent();
227
228 UFUNCTION()
229 void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
230 UFUNCTION()
231 void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
232
233

```

The constructor:

```

10 // ****
11 // Constructor
12 // ****
13 AVerletSim::AVerletSim()
14 {
15     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
16     PrimaryActorTick.bCanEverTick = true;
17
18     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
19     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
20
21     // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
22     TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));
23 }
24

```

In Begin play:

```

57
58     TextContainerIndex = -1;
59
60     TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
61     TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
62
63     if ( bEnableTriggerVolume )
64     {
65         TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AVerletSim::EnterVolume);
66         TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AVerletSim::ExitVolume);
67     }
68 }
69

```

Figure 250 – Adding to VerletSim (3 Images)

Addition of three new functions EnterVolume (a callback function called when the player enters the attached volume), OnEndSequence (called when the level sequence finishes) and ExitVolume (a callback called when the player leaves the volume).

```
339 // EnterVolume - Called when component enters collision volume
340 // ****
341 // ****
342 void AVerletSim::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
343 {
344     // Only allowed if overlapped by any kind of character
345     if ( Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
346         return;
347
348     if ( LevelSequence )
349     {
350         FLevelSequencePlaybackSettings Settings;
351
352         Settings.LoopCount = 0;
353
354         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
355         SequencePlayer->Play();
356     }
357 }
358
359 // ****
360 // OnEndSequenceEvent - Triggered when level sequence has finished playing
361 // ****
362 void AVerletSim::OnEndSequenceEvent()
363 {
364     SequencePlayer->Stop();
365     SequencePlayer = nullptr;
366
367     TextContainerIndex = -1;
368     AFinalProjectRyanTHUD::SetString();
369 }
370
371 // ****
372 // EnterVolume - Called when component exits collision volume
373 // ****
374 void AVerletSim::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
375 {
376     // Only allowed if overlapped by any kind of character
377     if ( Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
378         return;
379
380     if ( SequencePlayer )
381     {
382         SequencePlayer->Stop();
383         SequencePlayer = nullptr;
384     }
385     TextContainerIndex = -1;
386     AFinalProjectRyanTHUD::SetString();
387 }
388
```

*Figure 251*

Adding a Vector2D UPROPERTY to the VerletSim class for tiling of UV co-ordinates on the mesh.

```
212     UPROPERTY(EditAnywhere, Category = "Verlet Settings|MeshComponent") MeshComponent = nullptr;
213
214     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
215         UMaterial* MeshMaterial = nullptr;
216
217     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
218         FVector2D UVTileScale = FVector2D(1.0f, 1.0f);
219
220     float CollisionSphereSize = 2.0f;
221
222     TArray<VerletConstraint> ConstraintList;
223
224     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
225         TArray<FVerletAttachment> AttachmentList;
```

*Figure 252*

Adding code to rope to use UVTileScale.

```
203     // Build the vertex list
204
205     for (int i = 0; i < NumVerts; i++)
206     {
207         float Rad = 0.0f;
208         float RadDelta = (PI*2.0f) / float(RopeSegments); // Circumference divided by the number of segments (In Radians)
209         for ( int s = 0 ; s < RopeSegments ; s++ )
210         {
211             FVector Offset;
212             Offset.X = cosf( Rad ) * RopeThickness;
213             Offset.Y = sinf( Rad ) * RopeThickness;
214             Offset.Z = 0.0f;                                // Always on horizontal plane
215
216             FVector Pos = (Verts[i].Pos + Offset);
217
218             // DrawDebugLine(GetWorld(), Verts[i].Pos, Pos, FColor::Emerald, true, 20.0f, 0, 2.0f);           // Show this verts segment offsets for 20 seconds
219
220             Vertices.Add( Pos );
221
222             FVector Normal = Offset;
223             Normal.Normalize();
224             Normals.Add( Normal );
225
226             FVector2D UV = FVector2D(s,i);
227             UV.X /= float(RopeSegments);          // Horizontal: 0.0 to 1.0 mapped around the rope
228             UV.Y /= float(NumVerts);              // Vertical: 0.0 to 1.0 from top of rope to bottom
229
230             UV *= UVTileScale;
231
232             UV0.Add(UV);
233
234             Rad += RadDelta;                  // Next segment in radians
235         }
236     }
237 }
```

Figure 253

Adding code to cloth to use UVTileScale.

```
434     // Build the vertex list
435
436     for (int z = 0; z < NumberOfVertsH; z++)
437     {
438         for (int y = 0; y < NumberOfVertsW; y++)
439         {
440             Vertices.Add( Verts[y][z].Pos );
441
442             Normals.Add( FVector(1,0,0) );
443
444             FVector2D UV = FVector2D(y,z);
445             UV.X /= float(NumberOfVertsW);
446             UV.Y /= float(NumberOfVertsH);
447
448             UV *= UVTileScale;
449
450             UV0.Add(UV);
451         }
452     }
```

Figure 254

## Improved ResolveVertCollision for static mesh collision.

```
326 // ****
327 // ResolveVertCollision - Handles Collision For Verts
328 // ****
329 bool AVerletSim::ResolveVertCollision(Verlet& Vert)
330 {
331     if ( Vert.Flags )
332         return false;                                // Attached vert, so don't perform collision.
333
334     bool bCollided = false;
335
336     //-----
337     // Collision with static objects (Spheres only)
338     //-----
339     if ( bEnableStaticCollision )
340     {
341         FHitResult HitResult;
342         FCollisionQueryParams Params;
343         FCollisionObjectQueryParams ObjParams;
344         Params.AddIgnoredComponent( MeshComponent );           // Dont collide with self
345
346         ObjParams.AddObjectTypeToQuery( ECC_WorldStatic );
347         ObjParams.AddObjectTypeToQuery( ECC_WorldDynamic );
348         ObjParams.AddObjectTypeToQuery( ECC_PhysicsBody );
349
350         FVector RayStart = Vert.PreviousPos;
351         FVector RayEnd = Vert.Pos;
352
353         if (GetWorld()->LineTraceSingleByObjectType( HitResult, RayStart, RayEnd, ObjParams, Params))
354         {
355             if ( HitResult.Actor.IsValid() && HitResult.Actor->GetClass() != AMyStaticMesh::StaticClass() )
356             {
357                 float StandOff = 1.0f;          // Collision stand off distance
358
359                 Vert.Pos        = HitResult.ImpactPoint + HitResult.ImpactNormal * StandOff;    // Move out from impact a bit
360                 Vert.PreviousPos = Vert.Pos;
361
362                 if ( HitResult.GetComponent() )
363                 {
364                     // If it's movable, add an impulse to it
365                     if ( HitResult.GetComponent()->IsSimulatingPhysics() && HitResult.GetComponent()->Mobility == EComponentMobility::Movable )
366                     {
367                         FVector Impulse = -HitResult.ImpactNormal * 40.0f;
368                         HitResult.GetComponent()->AddImpulseAtLocation( Impulse, HitResult.ImpactPoint );
369                     }
370                 }
371                 bCollided = true;
372             }
373         }
374     }
375     return bCollided;
376 }
377
```

Figure 255

Improved ResolveVertCollision. Added simple sphere collision for dynamic objects.

```

129     bool AVerletSim::ResolveVertCollision(Verlet& Vert)
130     {
131         if ( Vert.Flags )
132             return false; // Attached vert, so don't perform collision.
133
134         bCollided = false;
135
136         //-----
137         // Collision with static objects (Spheres only)
138         //-----
139         if ( bEnableStaticCollision )
140         {
141             FHitResult HitResult;
142             FCollisionQueryParams Params;
143             FCollisionObjectQueryParams ObjParams;
144             Params.AddignoredComponent( MeshComponent );
145
146             ObjParams.AddObjectTypeToQuery( ECollisionChannel::ECC_WorldStatic );
147             ObjParams.AddObjectTypeToQuery( ECollisionChannel::ECC_WorldDynamic );
148             ObjParams.AddObjectTypeToQuery( ECollisionChannel::ECC_PhysicsBody );
149
150             FVector RayStart = Vert.PreviousPos;
151             FVector RayEnd = Vert.Pos;
152
153             if ( GetWorld() ->LineTraceSingleByObjectType( HitResult, RayStart, RayEnd, ObjParams, Params ) )
154             {
155                 if ( HitResult.Actor.IsValid() && HitResult.Actor->GetClass() != AMyStaticMesh::StaticClass() )
156                 {
157                     float StandOff = 1.0f; // Collision stand off distance
158
159                     Vert.Pos = HitResult.ImpactPoint + HitResult.ImpactNormal * StandOff; // Move out from impact a bit
160                     Vert.PreviousPos = Vert.Pos;
161
162                     if ( HitResult.GetComponent() )
163                     {
164                         // If it's movable, add an impulse to it
165                         if ( Hitresult.GetComponent() ->IsSimulatingPhysics() && HitResult.GetComponent()->Mobility == EComponentMobility::Movable )
166                         {
167                             FVector Impulse = -HitResult.ImpactNormal * 40.0f;
168                             HitResult.GetComponent()->AddImpulseAtLocation( Impulse, HitResult.ImpactPoint );
169                         }
170                     }
171                 }
172             }
173         }
174
175         //-----
176         // Collision with dynamic objects (Spheres only)
177         //-----
178         for ( int i = 0 ; i < AMyStaticMesh::MeshList.Num() ; i++ )
179         {
180             AMyStaticMesh *MyMesh = AMyStaticMesh::MeshList[i];
181
182             FVector SpherePos = MyMesh->GetActorLocation();
183
184             float Dist = FVector::Dist( SpherePos, Vert.Pos );
185
186             if ( Dist <= MyMesh->SphereRadius )
187             {
188                 DrawDebugSphere( GetWorld(), Vert.Pos, 10.0f, 8, FColor::White );
189
190                 FVector Dir = Vert.Pos - SpherePos; // Get vector from sphere centre to the vert position
191                 Dir.Normalize(); // Normalise it
192
193                 FVector ImpactPoint = SpherePos + Dir * MyMesh->SphereRadius; // Calculate impact point by moving out from the centre of sphere towards the vert by the sphere radius.
194                 Vert.Pos = ImpactPoint + Dir;
195                 Vert.PreviousPos = Vert.Pos;
196
197                 bCollided = true;
198             }
199         }
200     }
201
202     return bCollided;
203 }

```

Figure 256

Added Constraint list to VerletSim class.

```

197     float Timer = 0.0f; // General timer for animation debug stuff
198
199     AWindSim* Wind = nullptr;
200
201     UPrimitiveComponent* LastAddedComponent = nullptr;
202
203     UProceduralMeshComponent* MeshComponent = nullptr;
204
205     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
206         UMaterial* MeshMaterial = nullptr;
207
208     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
209         FVector2D UVTileScale = FVector2D(1.0f, 1.0f);
210
211     TArray<VerletConstraint> ConstraintList;
212
213     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
214         FVerletPhysicsSettings PhysicsSettings;

```

Figure 257

Added a function to allow derived classes (rope and cloth) to add constraint to the list (VerletSim class)

```

108 // ****
109 // AVerletSim - Base class for all verlet simulation classes
110 // ****
111 UCLASS()
112 class FINALPROJECTRYANT_API AVerletSim : public AActor
113 {
114     GENERATED_BODY()
115
116     public:
117         // Sets default values for this actor's properties
118         AVerletSim();
119
120         // Called when the game starts or when spawned
121         virtual void BeginPlay() override;
122
123         // Called every frame
124         virtual void Tick(float DeltaSeconds) override;
125
126         float GetParticleMass() { return ParticleMass; }
127
128     void CreateConstraint(Verlet* V1, Verlet* V2, eVerletConstraintType TypeIn)
129     {
130         VerletConstraint Constraint;
131
132         Constraint.Type = TypeIn;
133
134         Constraint.V1 = V1;
135         Constraint.V2 = V2;
136         Constraint.ConstraintDist = FVector::Dist( V1->Pos, V2->Pos );
137         ConstraintList.Add(Constraint);
138     }
139
140     protected:
141         // Mass for each vertex of the rope
142         UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics", meta = (ClampMin = 0.0f))
143             float ParticleMass = 5.0f;
144

```

Figure 258

Changed rope class to constrain using the new list.

```

126 // ****
127 // Constrain - Maintain equal distance between all verts
128 // ****
129 void ARopeSim::Constrain()
130 {
131     SCOPE_CYCLE_COUNTER(STAT_RopeConstrain);
132
133     //for (int i = 0; i < NumVerts - 1; i++)
134     //    ConstrainVert(Verts[i], Verts[i + 1], EdgeLength);
135
136     //for (int i = NumVerts - 1; i >= 1; i--)
137     //    ConstrainVert(Verts[i], Verts[i - 1], EdgeLength);
138
139     for (int i = 0; i < ConstraintList.Num() ; i++)
140     {
141         ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
142     }
143
144

```

Figure 259

Added a function rope to add all constraints to the constrain list.

```
365 // ****
366 // BuildConstraints - Build constraints for all verts
367 // ****
368 void ARopeSim::BuildConstraints()
369 {
370     for (int i = 0; i < NumVerts-1; i++)
371         CreateConstraint(&Verts[i], &Verts[i+1], eVerletConstraintType::Normal );
372 }
373
```

Figure 260

Added a function cloth to add all constraints to the constrain list.

```
178 // ****
179 // Constrain - Maintain equal distance between all verts
180 // ****
181 void AClothSim::Constrain()
182 {
183     for (int i = 0; i < ConstraintList.Num() ; i++)
184     {
185         ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
186     }
187 }
188
```

Figure 261

Added a function cloth to add all constraints to it's constrain list in a grid form.

```
564 // ****
565 // BuildConstraints - Build constraints for all verts
566 // ****
567 void AClothSim::BuildConstraints()
568 {
569     // Connect neighbouring verts with constraints (right, down, diagonal right, Diagonon left).
570     for (int x = 0; x<NumberOfVertsW; x++)
571     {
572         for (int y = 0; y<NumberOfVertsH; y++)
573         {
574             if (x<NumberOfVertsW - 1)
575                 CreateConstraint(&Verts[x][y], &Verts[x + 1][y], eVerletConstraintType::Normal);
576
577             if (y<NumberOfVertsH - 1)
578                 CreateConstraint(&Verts[x][y], &Verts[x][y + 1], eVerletConstraintType::Normal);
579
580             if (x<NumberOfVertsW - 1 && y<NumberOfVertsH - 1)
581             {
582                 CreateConstraint(&Verts[x][y], &Verts[x + 1][y + 1], eVerletConstraintType::Diagonal);
583                 CreateConstraint(&Verts[x + 1][y], &Verts[x][y + 1], eVerletConstraintType::Diagonal);
584             }
585         }
586     }
587
588     // Connect neighbouring verts with secondary constraints (right, down, diagonal right, Diagonon left).
589     // These span 2 cells, so they also constrain the curvature of the cloth surface too
590     for (int x = 0; x<NumberOfVertsW; x++)
591     {
592         for (int y = 0; y<NumberOfVertsH; y++)
593         {
594             if (x<NumberOfVertsW - 2)
595                 CreateConstraint(&Verts[x][y], &Verts[x + 2][y], eVerletConstraintType::NormalSecondary);
596
597             if (y<NumberOfVertsH - 2)
598                 CreateConstraint(&Verts[x][y], &Verts[x][y + 2], eVerletConstraintType::NormalSecondary);
599
600             if (x<NumberOfVertsW - 2 && y<NumberOfVertsH - 2)
601             {
602                 CreateConstraint(&Verts[x][y], &Verts[x + 2][y + 2], eVerletConstraintType::DiagonalSecondary);
603                 CreateConstraint(&Verts[x + 2][y], &Verts[x][y + 2], eVerletConstraintType::DiagonalSecondary);
604             }
605         }
606     }
607 }
```

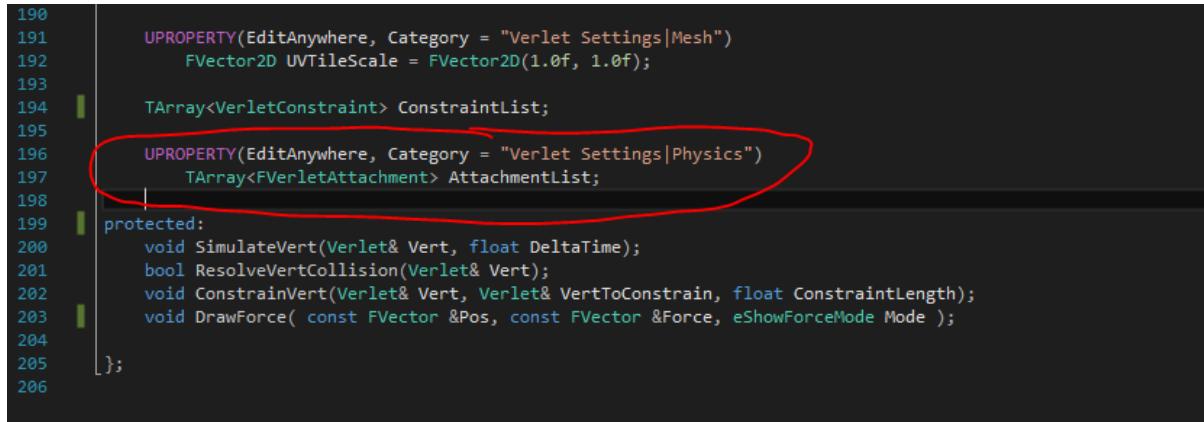
Figure 262

Added new dynamic attach structure to VerletSim class.

```
39 // ****
40 // FVerletAttachment - For adding dynamic attach/lock points to rope/cloth
41 // ****
42 USTRUCT()                                     // Tell Unreal Editor that this structure is exposed for editing in its details panels
43 struct FVerletAttachment
44 {
45     GENERATED_BODY()
46
47     UPROPERTY(EditAnywhere, Category = "Verlet Settings" )
48     bool        bRelativeToActor      = true;
49
50     UPROPERTY(EditAnywhere, Category = "Verlet Settings", Meta = (MakeEditWidget = true))
51     FVector    AttachPos          = FVector(0.0f,0.0f,0.0f);
52
53     UPROPERTY(EditAnywhere, Category = "Verlet Settings" )
54     int32      VertIndex          = 0;
55
56 };
```

Figure 263

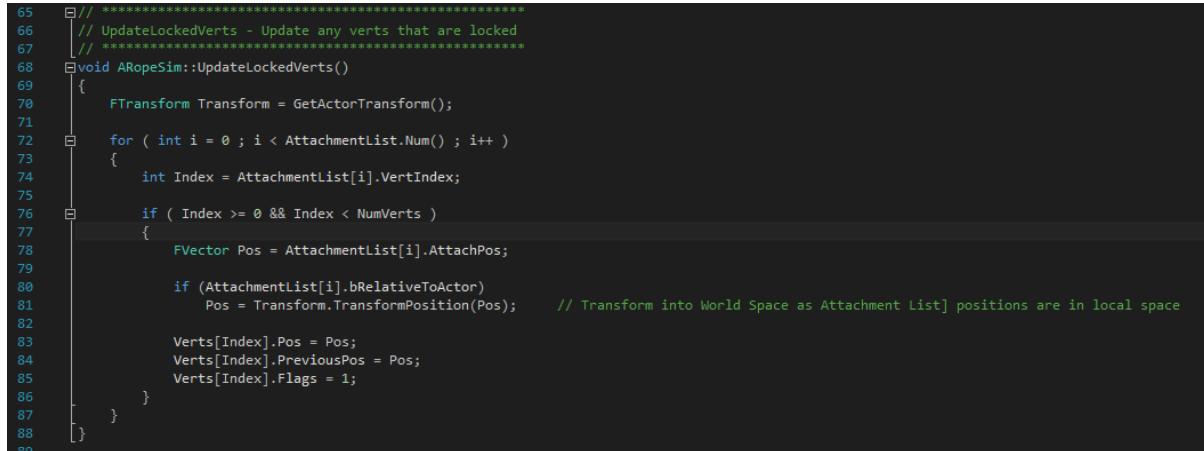
Added a list to contain dynamic attach points to VerletSim class.



```
190     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
191         FVector2D UVTileScale = FVector2D(1.0f, 1.0f);
192
193     TArray<VerletConstraint> ConstraintList;
194
195     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
196     TArray<FVerletAttachment> AttachmentList;
197
198 protected:
199     void SimulateVert(Verlet& Vert, float DeltaTime);
200     bool ResolveVertCollision(Verlet& Vert);
201     void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
202     void DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode );
203
204
205 };
206
```

Figure 264

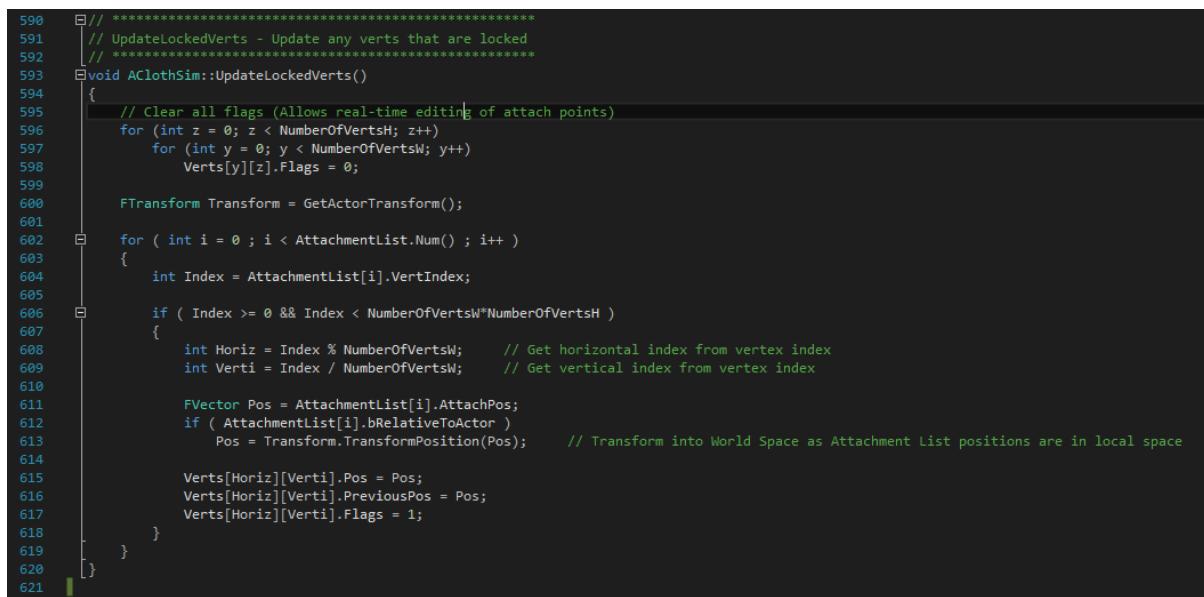
Updated Rope UpdateLockedVerts function to utilise the new list



```
65 // ****
66 // UpdateLockedVerts - Update any verts that are locked
67 // ****
68 void ARopeSim::UpdateLockedVerts()
69 {
70     FTransform Transform = GetActorTransform();
71
72     for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
73     {
74         int Index = AttachmentList[i].VertIndex;
75
76         if ( Index >= 0 && Index < NumVerts )
77         {
78             FVector Pos = AttachmentList[i].AttachPos;
79
80             if ( AttachmentList[i].bRelativeToActor )
81                 Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List] positions are in local space
82
83             Verts[Index].Pos = Pos;
84             Verts[Index].PreviousPos = Pos;
85             Verts[Index].Flags = 1;
86         }
87     }
88 }
```

Figure 265

Updated Cloth UpdateLockedVerts function to utilise the new list.



```
590 // ****
591 // UpdateLockedVerts - Update any verts that are locked
592 // ****
593 void AClothSim::UpdateLockedVerts()
594 {
595     // Clear all flags (Allows real-time editing of attach points)
596     for ( int z = 0; z < NumberOfVertsH; z++ )
597         for ( int y = 0; y < NumberOfVertsW; y++ )
598             Verts[y][z].Flags = 0;
599
600     FTransform Transform = GetActorTransform();
601
602     for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
603     {
604         int Index = AttachmentList[i].VertIndex;
605
606         if ( Index >= 0 && Index < NumberOfVertsW*NumberOfVertsH )
607         {
608             int Horiz = Index % NumberOfVertsW; // Get horizontal index from vertex index
609             int Verti = Index / NumberOfVertsW; // Get vertical index from vertex index
610
611             FVector Pos = AttachmentList[i].AttachPos;
612             if ( AttachmentList[i].bRelativeToActor )
613                 Pos = Transform.TransformPosition(Pos); // Transform into World Space as Attachment List positions are in local space
614
615             Verts[Horiz][Verti].Pos = Pos;
616             Verts[Horiz][Verti].PreviousPos = Pos;
617             Verts[Horiz][Verti].Flags = 1;
618         }
619     }
620 }
```

Figure 266

This is how the Attachments appear in the editor world. Each point can be moved with its own widget.

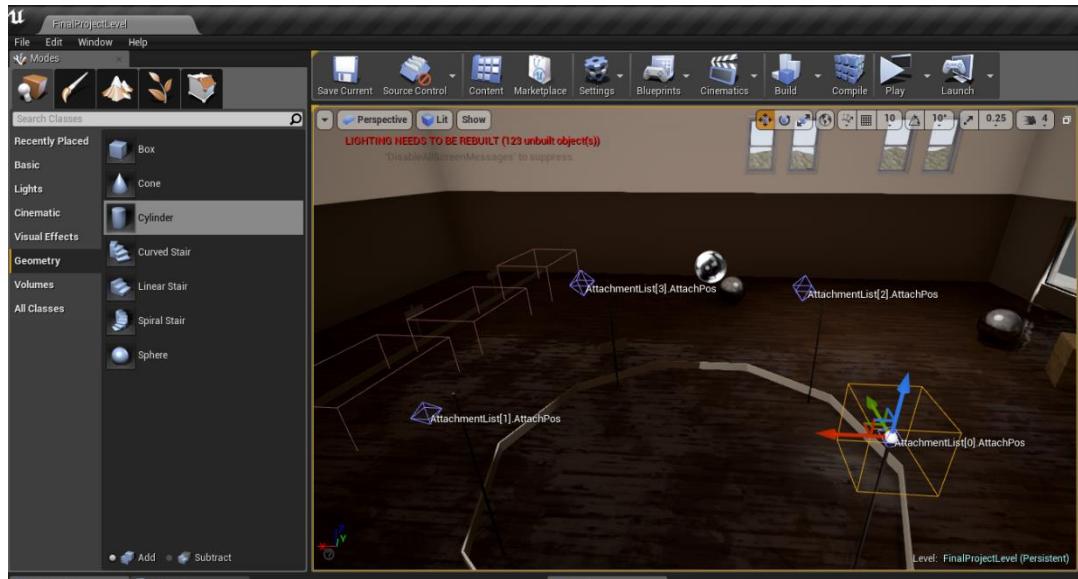


Figure 267

This is how the Attachments appear in the editor's details panel. Each point can be edited. New points can be added.

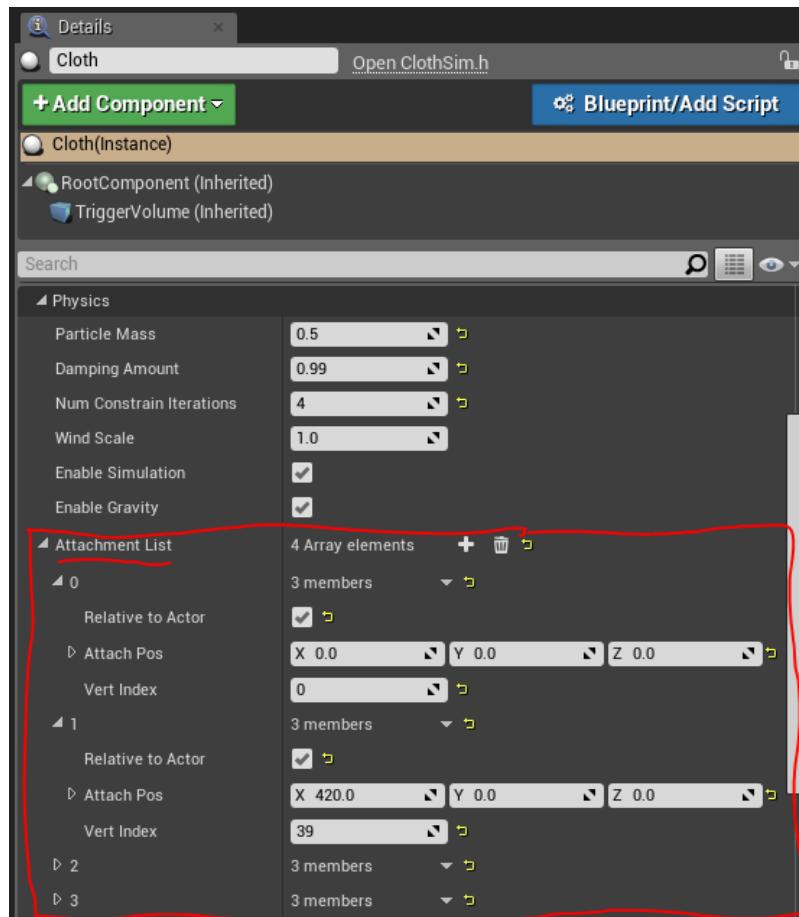
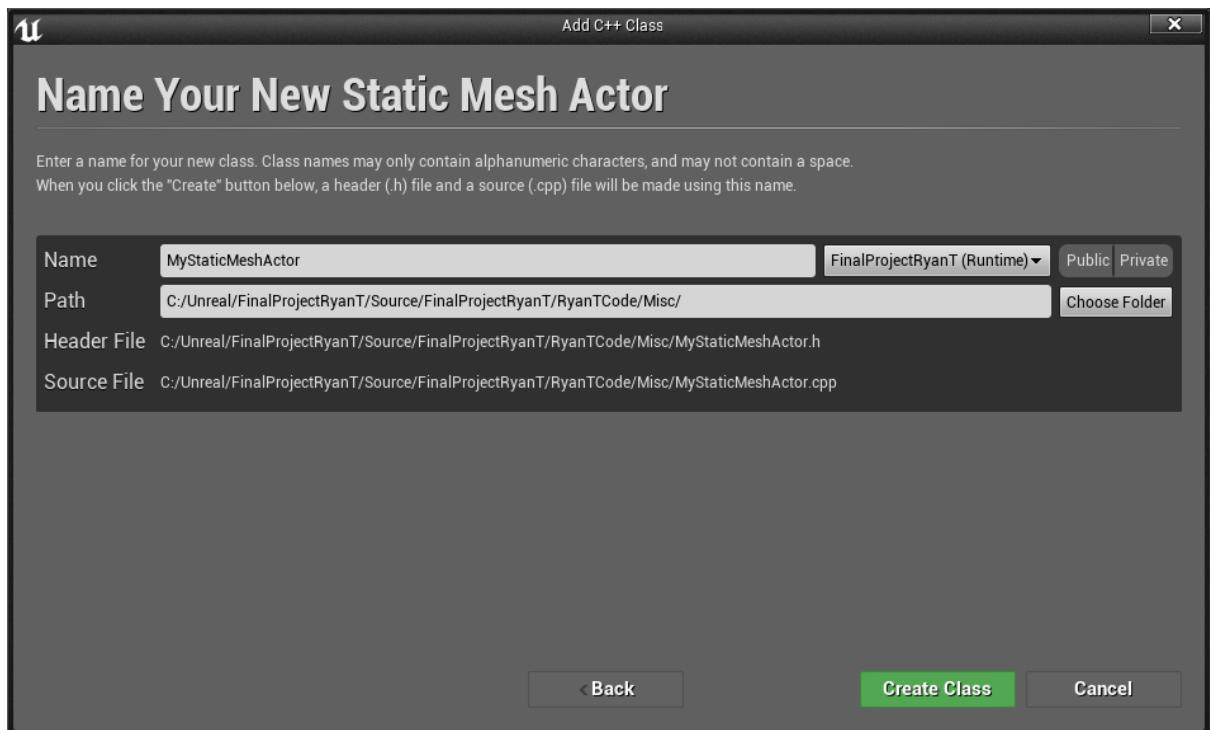
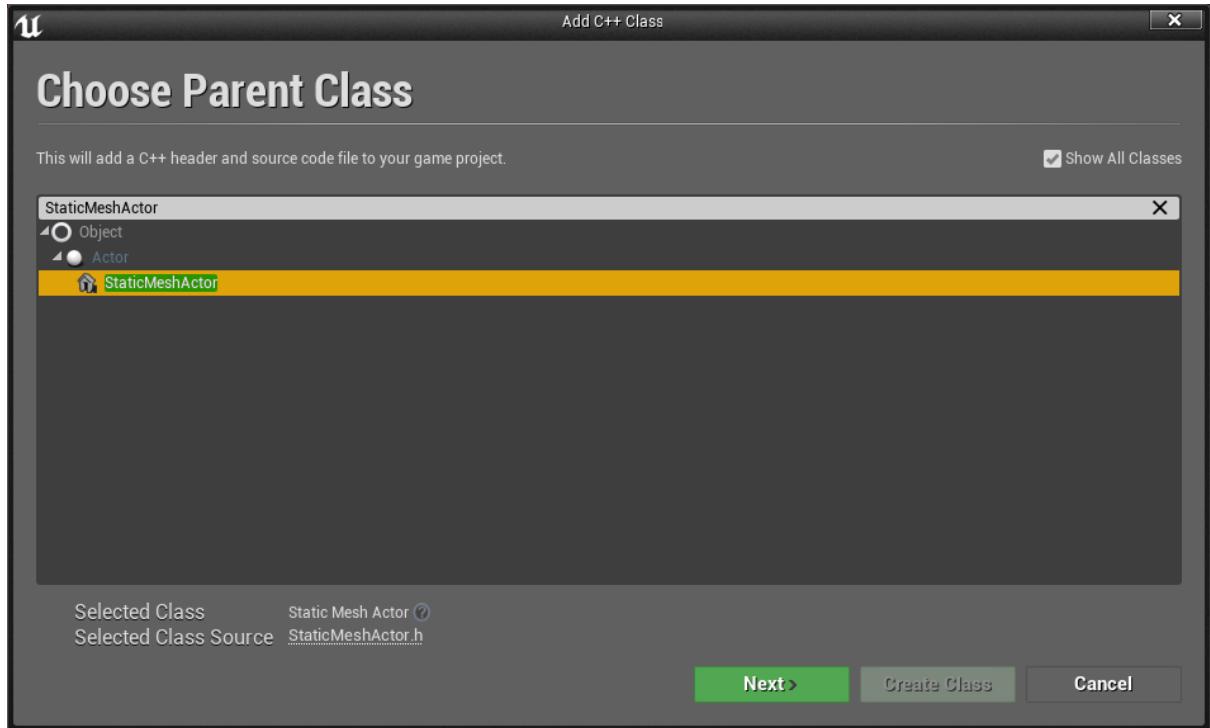


Figure 268

The next four screenshots are creating a new MyStaticMeshActor class derived from the Unreal class StaticMeshActor. This will be used for objects that require simple sphere collision with the cloth and ropes.



```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.  
2  
3 #pragma once  
4  
5 #include "Engine/StaticMeshActor.h"  
6 #include "MyStaticMesh.generated.h"  
7  
8 /**  
9 *  
10 */  
11 UCLASS()  
12 class FINALPROJECTRYANT_API AMyStaticMesh : public AStaticMeshActor  
13 {  
14     GENERATED_BODY()  
15 };  
16  
17
```

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.  
2  
3 #include "FinalProjectRyanT.h"  
4 #include "DrawDebugHelpers.h"  
5 #include "MyStaticMesh.h"  
6  
7 | |
```

Figure 269 – Making New Actor (4 Images)

The next two screenshots show adding Construct, BeginPlay, EndPlay and Tick to MyStaticMeshActor. Also added static list to contain all instances to use for collision in code later .

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "Engine/StaticMeshActor.h"
6 #include "MyStaticMesh.generated.h"
7
8 /**
9 *
10 */
11 UCLASS()
12 class FINALPROJECTRYANT_API AMyStaticMesh : public AStaticMeshActor
13 {
14     GENERATED_BODY()
15
16 public:
17     // Sets default values for this actor's properties
18     AMyStaticMesh();
19
20     // Called when the game starts or when spawned
21     virtual void BeginPlay() override;
22     virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;
23
24     // Called every frame
25     virtual void Tick(float DeltaSeconds) override;
26
27     UPROPERTY(EditAnywhere, Category = "MyStaticMesh Settings" )
28         float SphereRadius = 52.0f;
29
30     static TArray<AMyStaticMesh*> MeshList;
31 };
32
```

```

6     TArray<AMyStaticMesh*> AMyStaticMesh::MeshList;      // Static list that holds all 'AMyStaticMesh' instances. Used by collision.
7
8     // *****
9     // Constructor
10    // *****
11
12    AMyStaticMesh::AMyStaticMesh()
13    {
14        // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
15        PrimaryActorTick.bCanEverTick = true;
16    }
17
18    // *****
19    // Tick - Called every frame
20    // *****
21    void AMyStaticMesh::Tick(float DeltaTime)
22    {
23        Super::Tick(DeltaTime);
24    }
25
26    // *****
27    // BeginPlay - Called when the game starts or when spawned
28    // *****
29    void AMyStaticMesh::BeginPlay()
30    {
31        Super::BeginPlay();
32
33        MeshList.Add(this);      // Add to the mesh list
34    }
35
36    // *****
37    // EndPlay - Called when the game ends or when deleted
38    // *****
39    void AMyStaticMesh::EndPlay(const EEndPlayReason::Type EndPlayReason)
40    {
41        Super::EndPlay(EndPlayReason);
42
43        MeshList.Remove(this); // Remove from the mesh list
44    }
45

```

Figure 270 – Adding Functions (2 Images)

Added a ProceduralMeshComponent to VerletSim class plus a material to use for it.

```

195    float RenderModeTimer = 0.0f;
196    int RenderModeIndex = 0;
197
198    float Timer = 0.0f;           // General timer for animation debug stuff
199
200    AWindSim* Wind = nullptr;
201
202    UPrimitiveComponent* LastAddedComponent = nullptr;
203
204    UProceduralMeshComponent* MeshComponent = nullptr;
205
206    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
207        UMaterial* MeshMaterial = nullptr;
208
209
210    // *****
211    // BeginPlay - Called when the game starts or when spawned
212    // *****
213
214    void AVerletSim::BeginPlay()
215    {
216        Super::BeginPlay();
217
218        // Finds the first wind source object in the level
219        if (!Wind)
220        {
221            // GetAllActorsOfClass Searches the world and returns a list of all objects of the type required.
222            // In this case we are looking for all AWindSim objects.
223            TArray<Actor*> WindList;
224            UGameplayStatics::GetAllActorsOfClass(GetWorld(), AWindSim::StaticClass(), WindList);
225
226            if (WindList.Num())
227                Wind = Cast<AWindSim>(WindList[0]); // Use the first one in the list
228
229            MeshComponent = NewObject<UProceduralMeshComponent>(this, NAME_None);
230            MeshComponent->RegisterComponent();
231            MeshComponent->SetWorldTransform(FTransform()); // Set transform to Position(0,0,0) and Rotation(0,0,0)
232
233            // KeepWorldTransform means that the component is not relative to the AVerletSim actor, it has its own worldspace position
234            MeshComponent->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);
235            LastAddedComponent = MeshComponent;
236
237        }
238
239    }

```

Figure 271 – Additions to VerletSim (2 Images)

Added BuildMesh function to rope that initialises the procedural mesh component from the simulated vertices.

```

189     // ****
190     // BuildMesh - Build the textured mesh
191     // ****
192     void AROpeSim::BuildMesh()
193     {
194         MeshComponent->ClearAllMeshSections();      // Clear any old meshes
195
196         TArray< FVector>           Vertices;
197         TArray< int32>             Triangles;
198         TArray< FVector>           Normals;
199         TArray< FVector2D>          UV0;
200         TArray< FColor>             VertexColor;
201         TArray< FProcMeshTangent>   VertexCTangent;
202
203         // Build the vertex list
204
205         for (int i = 0; i < NumVerts; i++)
206         {
207             float Rad = 0.0f;
208             float RadDelta = (PI*2.0f) / float(RopeSegments);    // Circumference divided by the number of segments (In Radians)
209             for (int s = 0 ; s < RopeSegments ; s++)
210             {
211                 FVector Offset;
212                 Offset.X = cosf( Rad ) * RopeThickness;
213                 Offset.Y = sinf( Rad ) * RopeThickness;
214                 Offset.Z = 0.0f;                                // Always on horizontal plane
215
216                 FVector Pos = (Verts[i].Pos + Offset);
217
218                 // DrawDebugLine(GetWorld(), Verts[i].Pos, Pos, FColor::Emerald, true, 20.0f, 0, 2.0f);      // Show this verts segment offsets for 20 seconds
219
220                 Vertices.Add( Pos );
221
222                 FVector Normal = Offset;
223                 Normal.Normalize();
224                 Normals.Add( Normal );
225
226                 FVector2D UV = FVector2D(s,i);
227                 UV.X /= float(RopeSegments);        // Horizontal: 0.0 to 1.0 mapped around the rope
228                 UV.Y /= float(NumVerts);           // Vertical: 0.0 to 1.0 from top of rope to bottom
229
230                 UV *= UVTileScale;
231
232                 UV0.Add(UV);
233
234                 Rad += RadDelta;                // Next segment in radians
235             }
236         }
237
238         // Build the triangle list. Each trianlge has 3 x Indices into the above vertex list.
239         //
240         //      0---1
241         //      | \ |
242         //      | \ |
243         //      | \ |
244         //      | \ |
245         //      | \ |
246         //      | \ |
247         //      | \ |
248         //      | \ |
249         //      | \ |
250         //      | \ |
251         //      | \ |
252         //      | \ |
253         //      | \ |
254         //      | \ |
255         //      | \ |
256         //      | \ |
257         //      | \ |
258         //      | \ |
259         //      | \ |
260         //      | \ |
261         //      | \ |
262         //      | \ |
263         bool bCreateCollision = false;
264         MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );
265
266         MeshComponent->SetMaterial(0,MeshMaterial);
267
268         // MeshComponent->SetWorldLocation( GetActorLocation() );
269         MeshComponent->SetWorldTransform( FTransform() );      // Set transform to ID (Zero Position & Zero Rotation) as all verts are in world space.
270     }

```

Figure 272

Added UpdateMesh function to rope that updates the procedural mesh component from the simulated vertices every frame.

```

273 // ****
274 // UpdateMesh - Update the textured mesh
275 // ****
276 void ARopeSim::UpdateMesh()
277 {
278     SCOPE_CYCLE_COUNTER(STAT_RopeUpdateMesh);
279
280     TArray <FVector> Vertices;
281     TArray <FVector> Normals;
282     TArray <Vector2D> UV0;
283     TArray <Color> Vertexcolor;
284     TArray <FProcMeshTangent> VertexTangent;
285
286     FVector SurfaceNormals[NumVerts*MAX_ROPE_SEGMENTS];
287
288     for (int i = 0; i < NumVerts; i++)
289     {
290         //FVector Up      = FVector( 0,0,1 );
291         //FVector Side   = FVector( 0,1,0 );
292         //FVector Forward = FVector( 1,0,0 );
293
294         FVector Up, Side, Forward;
295
296         // Get the vector pointing up the rope at this vert
297         if ( i < Numverts-1 )
298             Up = Verts[i].Pos - Verts[i+1].Pos;
299         else
300             Up = Verts[i-1].Pos - Verts[i].Pos;
301         Up.Normalize();
302
303         Side   = FVector::CrossProduct( Up, FVector(0,1,0) ); // Get the side vector (Perpendicular to the world Y Axis (Forward) )
304         Forward = FVector::CrossProduct( Up, Side ); // Get the forward vector (Perpendicular Side axis and Up axis)
305         Side   = FVector::CrossProduct( Up, Forward ); // Get the 'real' side vector (Perpendicular to Up and Forward)
306
307         float Rad = 0.0f;
308         float RadDelta = (PI*2.0f) / float(RopeSegments); // Circumference divided by the number of segments (In Radians)
309         for (int s = 0; s < RopeSegments; s++)
310         {
311 #if 0 //Inactive Preprocessor Block
312 #else
313             FVector Offset = Side * cosf(Rad);
314             Offset += Forward * sinf(Rad);
315
316             Offset.Normalize();
317
318             FVector Pos = (Verts[i].Pos + Offset * RopeThickness);
319
320             Vertices.Add(Pos);
321
322 #if 1
323             SurfaceNormals[(i*RopeSegments)+s] = Offset;
324
325             // DrawDebugLine(GetWorld(), Pos, Pos + SurfaceNormals[(i*RopeSegments)+s] * 20.0f, FColor::Cyan, false, -1.0f, 0, 2.0f);
326
327 #endif
328
329             Rad += RadDelta; // Next segment in radians
330         }
331     }
332
333
334
335     // Build the vertex normals from the surface normals
336     for (int i = 0; i < NumVerts; i++)
337     {
338         float Rad = 0.0f;
339         float RadDelta = (PI*2.0f) / float(RopeSegments); // Circumference divided by the number of segments (In Radians)
340         for (int s = 0; s < RopeSegments; s++)
341         {
342             FVector Normal(0.0f,0.0f,0.0f);
343
344             if ( i > 0 )
345                 Normal += SurfaceNormals[((i-1)*RopeSegments)+s];
346             if ( i < (Numverts-1) )
347                 Normal += SurfaceNormals[((i+1)*RopeSegments)+s];
348
349             if ( s > 0 )
350                 Normal += SurfaceNormals[(i*RopeSegments)+(s-1)];
351             if ( s < (RopeSegments-1) )
352                 Normal += SurfaceNormals[(i*RopeSegments)+(s+1)];
353
354             Normal.Normalize();
355             Normals.Add( Normal );
356
357             //FVector Pos = Vertices[(i*RopeSegments)+s] + GetActorLocation();
358             //DrawDebugLine(GetWorld(), Pos, Pos + Normal * 20.0f, FColor::Cyan, false, -1.0f, 0, 2.0f);
359         }
360     }
361
362     MeshComponent->UpdateMeshSection( 0, Vertices, Normals, UV0, Vertexcolor, VertexTangent );
363 }

```

Figure 273

Added BuildMesh function to Cloth that initialises the procedural mesh component from the simulated vertices.

```

399 // ****
400 // BuildMesh - Build the textured mesh
401 // ****
402 void AClothSim::BuildMesh()
403 {
404     //if ( RenderMode != eRenderMode::ProgressiveMesh )
405     //    return;
406
407     MeshComponent->ClearAllMeshSections();      // Clear any old meshes
408
409     TArray <FVector>           Vertices;
410     TArray <int32>             Triangles;
411     TArray <FVector>           Normals;
412     TArray < FVector2D>        UV0;
413     TArray < FColor >          VertexColor;
414     TArray < FProcMeshTangent > VertexCTangent;
415
416     // Build the vertex list
417     for (int z = 0; z < NumberOfVertsH; z++)
418     {
419         for (int y = 0; y < NumberOfVertsW; y++)
420         {
421             Vertices.Add( Verts[y][z].Pos );
422
423             Normals.Add( FVector(1,0,0) );
424
425             FVector2D UV = FVector2D(y,z);
426             UV.X /= float(NumberOfVertsW);
427             UV.Y /= float(NumberOfVertsH);
428
429             UV *= UVTileScale;
430
431             UV0.Add(UV);
432         }
433     }
434
435     // Build the triangle list. Each triangle has 3 x Indices into the above vertex list.
436     //
437     //    0---1
438     //    | \  |
439     //    |   \ |
440     //    |      \
441     //    |      \
442     //    W+0---W+1
443     //
444     for (int z = 0; z < NumberOfVertsH-1; z++)
445     {
446         for (int y = 0; y < NumberOfVertsW-1; y++)
447         {
448             int32 BaseIndex = y+(z*NumberOfVertsW);
449
450             Triangles.Add( BaseIndex + 0 );
451             Triangles.Add( BaseIndex + NumberOfVertsW );
452             Triangles.Add( BaseIndex + NumberOfVertsW + 1 );
453
454             Triangles.Add( BaseIndex + NumberOfVertsW+1 );
455             Triangles.Add( BaseIndex + 1 );
456             Triangles.Add( BaseIndex + 0 );
457         }
458     }
459
460     bool bCreateCollision = true;
461     MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );
462
463 //    MeshComponent->BodyInstance.SetCollisionProfileName( UCollisionProfile::PhysicsActor_ProfileName );
464 //    MeshComponent->BodyInstance.SetCollisionEnabled( ECollisionEnabled::Type::QueryAndPhysics );
465
466 //    MeshComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
467 //    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn, ECollisionResponse::ECR_Block);
468 //    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_Vehicle, ECollisionResponse::ECR_Block);
469 //    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_Worldstatic, ECollisionResponse::ECR_Block);
470 //    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldDynamic, ECollisionResponse::ECR_Block);
471 //    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_PhysicsBody, ECollisionResponse::ECR_Block);
472 //    MeshComponent->SetNotifyRigidBodyCollision(true);
473
474     MeshComponent->SetMaterial(0,MeshMaterial);
475 }
476

```

Figure 274

Added UpdateMesh function to Cloth that updates the procedural mesh component from the simulated vertices.

```

477  //*****
478  // UpdateMesh - Update the textured mesh
479  //*****
480 void AClothSim::UpdateMesh()
481 {
482     if ( RenderMode != eRenderMode::ProgressiveMesh )
483         return;
484
485     SCOPE_CYCLE_COUNTER(STAT_ClothUpdateMesh);
486
487     TArray < FVector > Vertices;
488     TArray < FVector > Normals;
489     TArray < FVector2D > UV0;
490     TArray < FColor > VertexColor;
491     TArray < FProcMeshTangent > VertexCTangent;
492
493     FVector SurfaceNormals[MaxVertsW][MaxVertsH];
494
495     for ( int z = 0; z < NumberOfVertsH; z++ )
496     {
497         for ( int y = 0; y < NumberOfVertsW; y++ )
498         {
499             // Add the vert to the list
500             Vertices.Add( Verts[y][z].Pos );
501
502             // Create a surface normal ready for next stage (Vertex Normal)
503             FVector Normal, V1, V2;
504             if ( y < (NumberOfVertsW-1) )
505                 V1 = Verts[y+1][z].Pos - Verts[y][z].Pos;
506             else
507                 V1 = Verts[y][z].Pos - Verts[y-1][z].Pos;
508
509             if ( z < (NumberOfVertsH-1) )
510                 V2 = Verts[y][z+1].Pos - Verts[y][z].Pos;
511             else
512                 V2 = Verts[y][z].Pos - Verts[y][z-1].Pos;
513
514             // Add the surface normal to a local list ready for the next stage
515             SurfaceNormals[y][z] = FVector::CrossProduct( V1, V2 );
516             SurfaceNormals[y][z].Normalize();
517         }
518     }
519
520     // Build the vertex normals from the surface normals
521     for ( int z = 0; z < NumberOfVertsH; z++ )
522     {
523         for ( int y = 0; y < NumberOfVertsW; y++ )
524         {
525             FVector Normal(0.0f,0.0f,0.0f);
526
527             if ( z > 0 )
528                 Normal += SurfaceNormals[y][z-1];
529             if ( z < (NumberOfVertsH-1) )
530                 Normal += SurfaceNormals[y][z+1];
531
532             if ( y > 0 )
533                 Normal += SurfaceNormals[y-1][z];
534             if ( y < (NumberOfVertsW-1) )
535                 Normal += SurfaceNormals[y+1][z];
536
537             Normal.Normalize();
538             Normals.Add( Normal );
539         }
540     }
541
542     MeshComponent->UpdateMeshSection( 0, Vertices, Normals, UV0, VertexColor, VertexCTangent );
543 }
544

```

Figure 275

Cloth with new procedural mesh added.

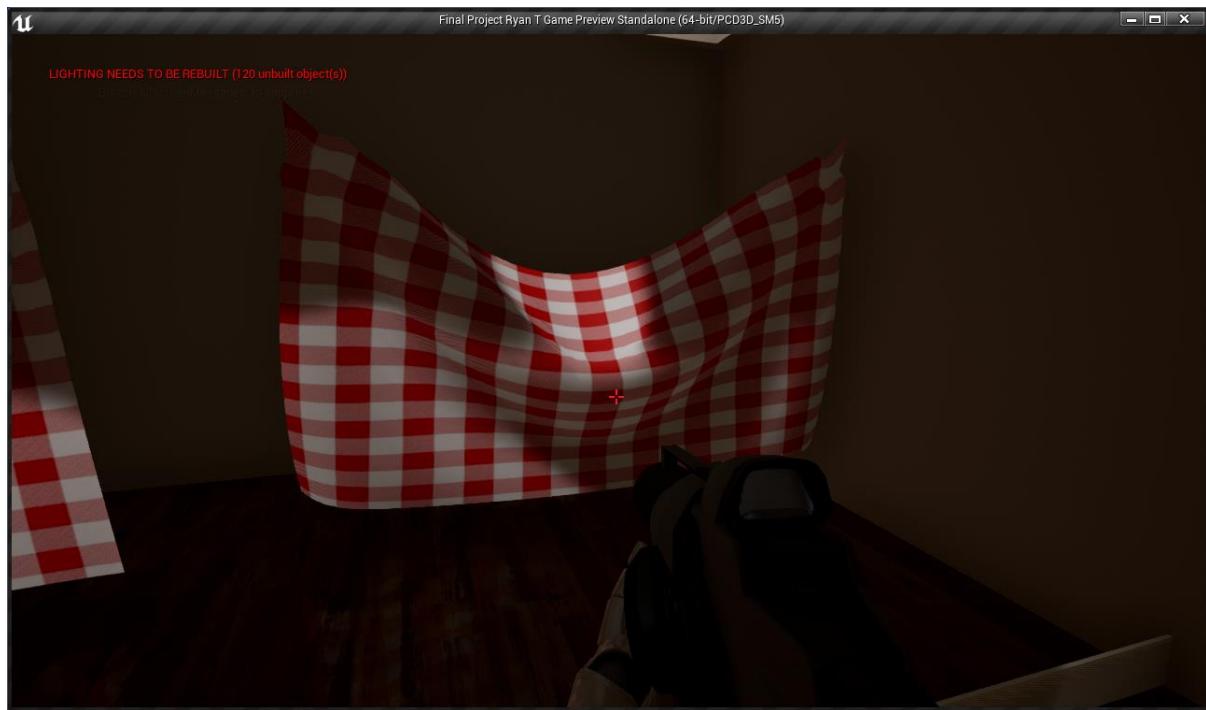


Figure 276

Rope with new procedural mesh added.



Figure 277

Added various UPROPERTY variables to verletSim class to enable level sequence to be able to control how rope and cloth is rendered (wireframe etc.).

```

136
137     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Collision")
138         bool bEnableStaticCollision = false;
139
140     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Physics")
141         float WindScale = 1.0f;
142
143     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
144         bool bEnableSimulation = true;
145
146     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Physics")
147         bool bEnableGravity = true;
148
149     // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
150     // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.
151
152     // Toggles display of debug draw lines
153     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings", meta = (DisplayName = "Enable Debug Draw"))
154         bool bEnableDebugDraw = false;
155
156     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings")
157         bool bShowLockedVerts = false;
158
159     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Debug Settings")
160         eShowForceMode ShowForcesMode = eShowForceMode::None;
161
162     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Debug Settings")
163         eVerletConstraintType DrawConstraintFilter = eVerletConstraintType::All;
164
165     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Render Mode")
166         eRenderMode RenderMode = eRenderMode::ProgressiveMesh;
167
168     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Render Mode")
169         bool bRenderModeIncremental = false;
170
171     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Render Mode")
172         float RenderModeIncrementalTime = 1.0f;
173
174     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
175         float EdgeLength = 10.0f;
176

```

Figure 278

New render function for cloth that uses the previously mentioned UPROPERTY variables to call the relevant render function.

```

351     // *****
352     // Render - Renders the cloth
353     // *****
354     void AClothSim::Render()
355     {
356         switch ( RenderMode )
357         {
358             case eRenderMode::None:
359                 break;
360
361             case eRenderMode::WireframeTriangles:
362             {
363                 DrawWireframeTriangles( bRenderModeIncremental );
364                 break;
365             }
366
367             case eRenderMode::WireframeSquares:
368             {
369                 DrawWireframeSquares( bRenderModeIncremental );
370                 break;
371             }
372
373             case eRenderMode::WireframeConstraints:
374             {
375                 DrawWireframeConstraints( bRenderModeIncremental );
376                 break;
377             }
378
379             case eRenderMode::WireframeSquaresAndConstraints:
380             {
381                 DrawWireframeSquares( false /*bRenderModeIncremental*/ );
382                 DrawWireframeConstraints( bRenderModeIncremental );
383                 break;
384             }
385
386             case eRenderMode::ProgressiveMesh:
387             {
388                 UpdateMesh();
389                 break;
390             }
391         }
392     }

```

Figure 279

New debug draw functions to be used during level sequenced demos.

```
187 // ****
188 // DrawWireframeTriangles - Draws the mesh as wireframe triangles
189 // ****
190 void AClothSim::DrawWireframeTriangles( bool bIncremental )
191 {
192     if ( bRenderModeIncremental )
193     {
194         int NumTriangles = (NumberOfVertsW-1)*(NumberOfVertsH-1) * 2;
195
196         if ( RenderModeIndex >= NumTriangles )
197             RenderModeIndex = 0;
198     }
199
200     FColor Colour = FColor::Green;
201
202     int Count = 0;
203     for (int z = 0; z < NumberOfVertsH - 1; z++)
204     {
205         for (int y = 0; y < NumberOfVertsW - 1; y++)
206         {
207             //A-0,0----0,1-B
208             //    | \   |
209             //    | \   |
210             //    | \   |
211             //    | \   |
212             //C-0,1----1,1-D
213
214             FVector& A = Verts[y][z].Pos;
215             FVector& B = Verts[y][z+1].Pos;
216             FVector& C = Verts[y+1][z].Pos;
217             FVector& D = Verts[y+1][z+1].Pos;
218
219             if ( bRenderModeIncremental && Count > RenderModeIndex )
220                 break;
221
222             // Triangle 1
223             DrawDebugLine(GetWorld(), A, B, Colour, false, -1.0f, 0, 1.0f);
224             DrawDebugLine(GetWorld(), B, D, Colour, false, -1.0f, 0, 1.0f);
225             DrawDebugLine(GetWorld(), D, A, Colour, false, -1.0f, 0, 1.0f);
226             Count++;
227
228             if ( bRenderModeIncremental && Count > RenderModeIndex )
229                 break;
230
231             // Triangle 2
232             DrawDebugLine(GetWorld(), A, D, Colour, false, -1.0f, 0, 1.0f);
233             DrawDebugLine(GetWorld(), D, C, Colour, false, -1.0f, 0, 1.0f);
234             DrawDebugLine(GetWorld(), C, A, Colour, false, -1.0f, 0, 1.0f);
235             Count++;
236         }
237     }
238 }
```

```

240 // ****
241 // DrawWireframeSquares - Draws the mesh as wireframe squares
242 // ****
243 void AClothSim::DrawWireframeSquares( bool bIncremental )
244 {
245     if ( bRenderModeIncremental )
246     {
247         if ( RenderModeIndex >= ((NumberOfVertsW-1)*(NumberOfVertsH-1)) )
248             RenderModeIndex = 0;
249     }
250
251     // For each vert, draw a debug line, alternating between the colours red and blue.
252     int Count = 0;
253     for (int z = 0; z < NumberOfVertsH - 1; z++)
254     {
255         bool bIsLastRow = z == (NumberOfVertsH - 2);
256
257         for (int y = 0; y < NumberOfVertsW - 1; y++)
258         {
259             if ( bRenderModeIncremental && Count > RenderModeIndex )
260                 break;
261
262             bool bIsLastColumn = y == (NumberOfVertsW - 2);
263
264             //A-0,0----1,0-B
265             //      | \   |
266             //      |  \  |
267             //      |   \ |
268             //      |    \| |
269             //C-0,1----1,1-D
270
271             FVector& A = Verts[y][z].Pos;
272             FVector& B = Verts[y+1][z].Pos;
273             FVector& C = Verts[y][z+1].Pos;
274             FVector& D = Verts[y+1][z+1].Pos;
275
276             FColor Colour = FColor::Green;
277
278             // Draw top line
279             DrawDebugLine(GetWorld(), A, B, Colour, false, -1.0f, 0, 1.0f);
280
281             if ( bRenderModeIncremental || bIsLastRow )
282             {
283                 // Draw Bottom line if we are drawing the last row
284                 DrawDebugLine(GetWorld(), C, D, Colour, false, -1.0f, 0, 1.0f);
285             }
286
287             // Colour = FColor::Red;
288
289             // Draw left line
290             DrawDebugLine(GetWorld(), A, C, Colour, false, -1.0f, 0, 1.0f);
291
292             if ( bRenderModeIncremental || bIsLastColumn )
293             {
294                 // Draw Right line if we are drawing the last column
295                 DrawDebugLine(GetWorld(), B, D, Colour, false, -1.0f, 0, 1.0f);
296             }
297             Count++;
298         }
299     }
300 }
301

```

```

302     // ****
303     // DrawWireframeConstraints - Draws all constraints
304     // ****
305     void AClothSim::DrawWireframeConstraints( bool bIncremental )
306     {
307         for ( int i = 0; i < ConstraintList.Num() ; i++ )
308         {
309             // Don't show diagonal constraints
310             if ( DrawConstraintFilter != eVerletConstraintType::All && ConstraintList[i].Type != DrawConstraintFilter )
311                 continue;
312
313             FColor Colour = FColor::Green;
314
315             float Length = (ConstraintList[i].V1->Pos - ConstraintList[i].V2->Pos).Size();
316
317             // Apply some modulation to the length
318
319             float Modulation = sinf(Timer*5.0f);           // Get modulating value from -1 to +1
320             Modulation += 1.0f;                            // Add 1 to put in range 0 to 2
321             Modulation *= 0.5f;                           // divide by 2 to put in range 0 to 1
322
323             Modulation *= 0.3f;                          // Change to range 0.7 to 1.0
324             Modulation += 0.7f;
325
326             Length *= Modulation;                      // Scale Length by the modulating value from 0.0 to 1.0
327
328 #if 1
329             // 2 arrows, outwards from line centre
330             FVector Centre = (ConstraintList[i].V1->Pos + ConstraintList[i].V2->Pos) * 0.5f;           // Centre of the line = the average of both positions
331
332             FVector Dir = (ConstraintList[i].V1->Pos - ConstraintList[i].V2->Pos);
333             Dir.Normalize();
334
335             FVector Offset = Dir * Length * 0.5f;
336
337             DrawDebugDirectionalArrow( GetWorld(), Centre, Centre + Offset, 7.0f, FColor::Magenta, false, 0.0f, 0, 1.0f );
338             DrawDebugDirectionalArrow( GetWorld(), Centre, Centre - Offset, 7.0f, FColor::Magenta, false, 0.0f, 0, 1.0f );
339 #else
340             // 1 arrow from vert 1 to vert 2
341             FVector Dir = (ConstraintList[i].V2->Pos - ConstraintList[i].V1->Pos);
342             Dir.Normalize();
343
344             FVector Offset = Dir * Length;
345
346             DrawDebugDirectionalArrow( GetWorld(), ConstraintList[i].V1->Pos, ConstraintList[i].V1->Pos + Offset, 7.0f, FColor::Magenta, false, 0.0f, 0, 1.0f );
347 #endif
348         }
349     }
350

```

Figure 280 – Debug Draw Functions(3 Images)

### Starting Scripting the Centre Demo.



Figure 281

Adding an instance of MyStaticMesh actor for cloth dynamic collision demo.

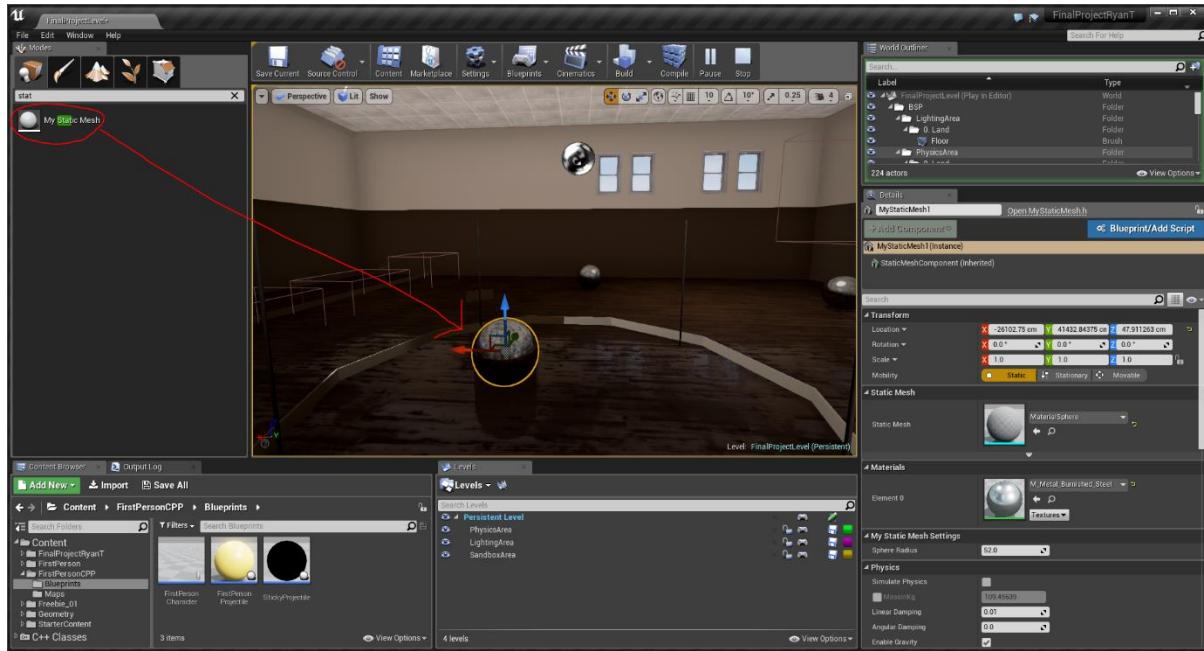


Figure 282

Position Cloths trigger box component to allow a level sequence to trigger on entry.

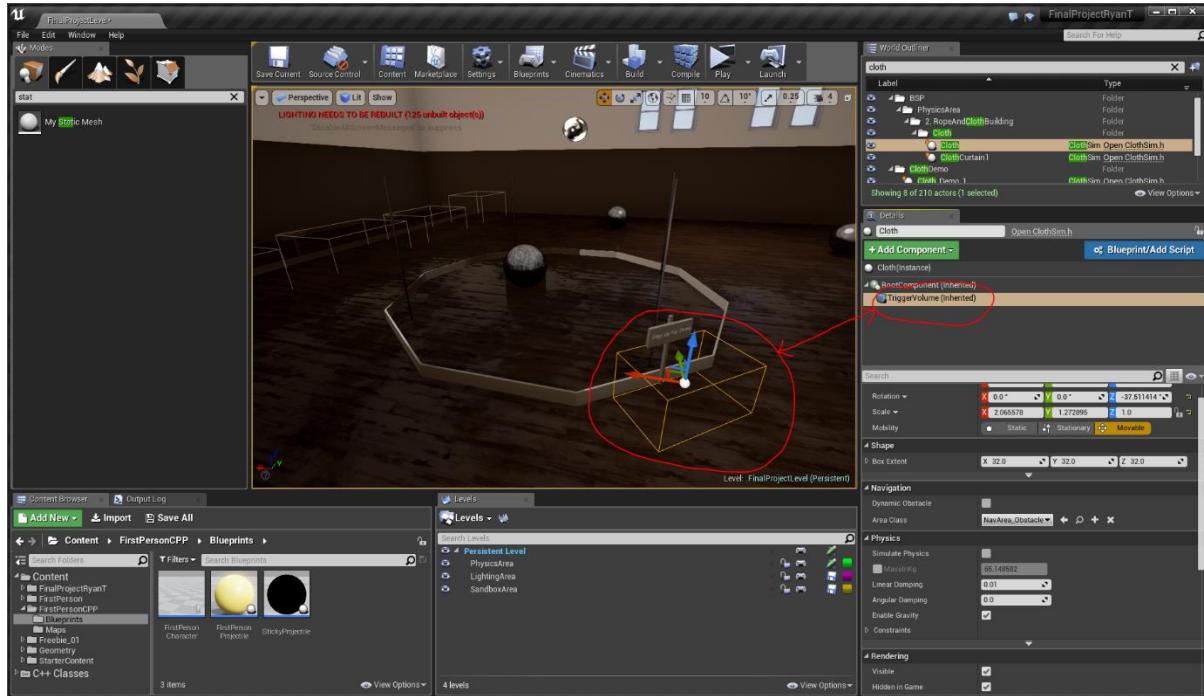


Figure 283

## Creating and naming an empty Level Sequence.

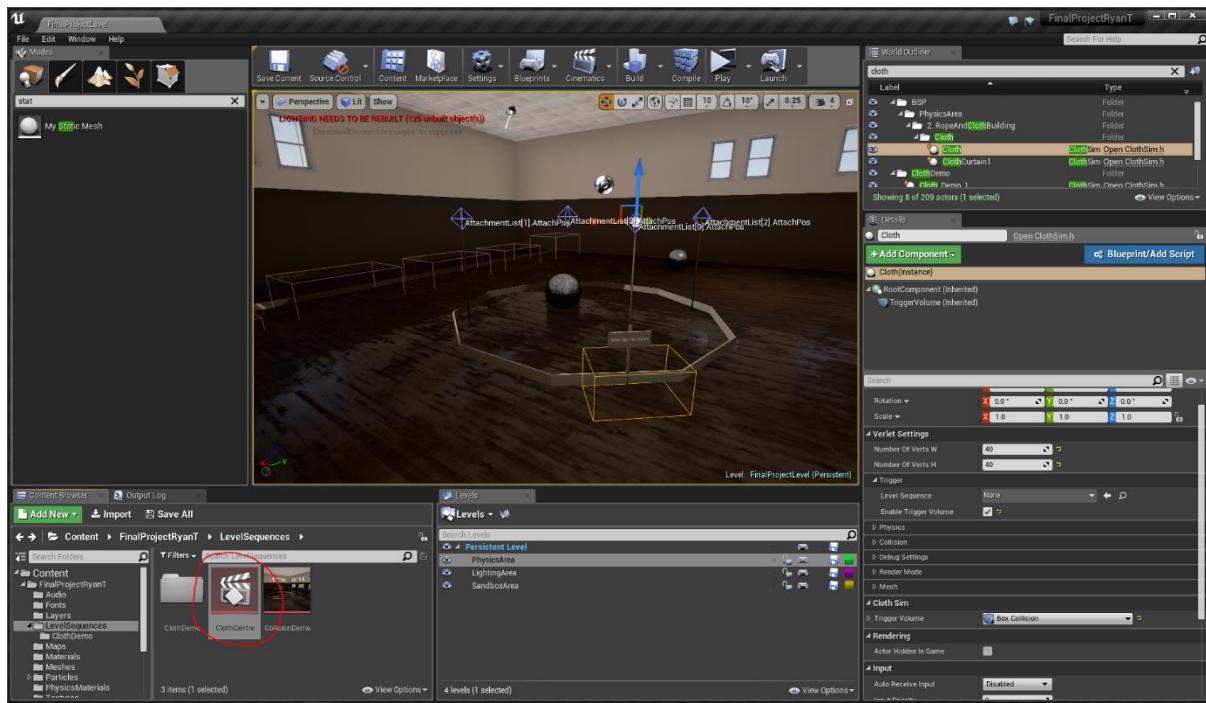


Figure 284

The empty Level Sequence.

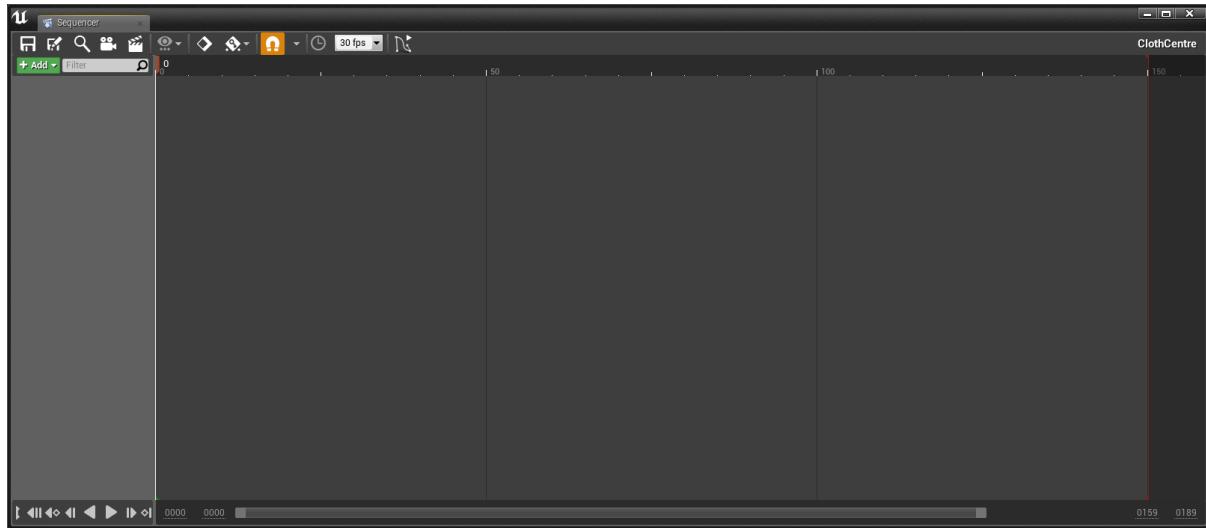


Figure 285

Adding the Level Sequence to the centre cloth object. This will now be triggered when player enters the volume.

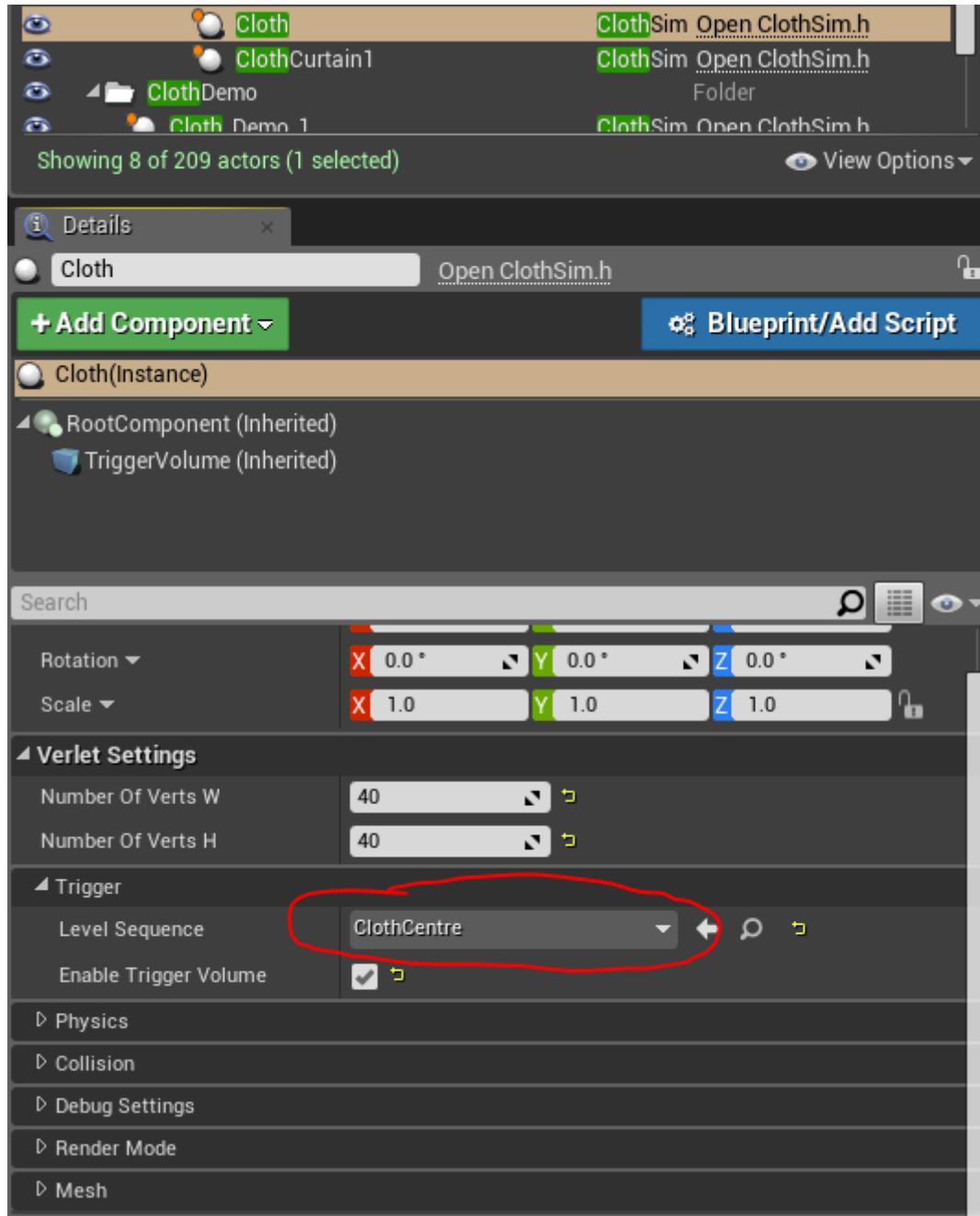


Figure 286

Adding the MyStaticMesh (Centre Sphere) to the Level Sequence.

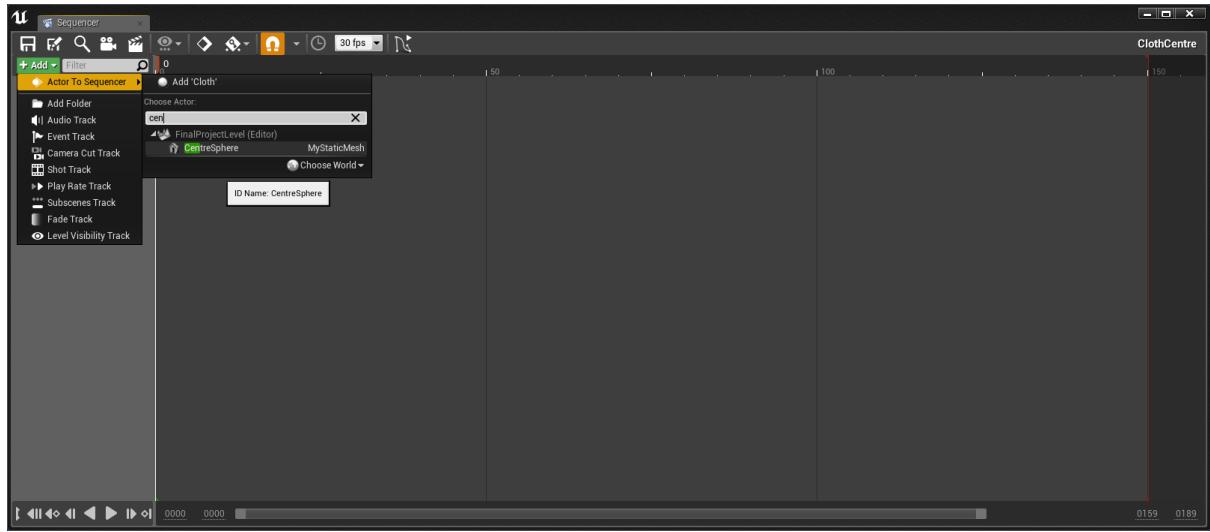


Figure 287

Added with a transform track to that it can be moved about.

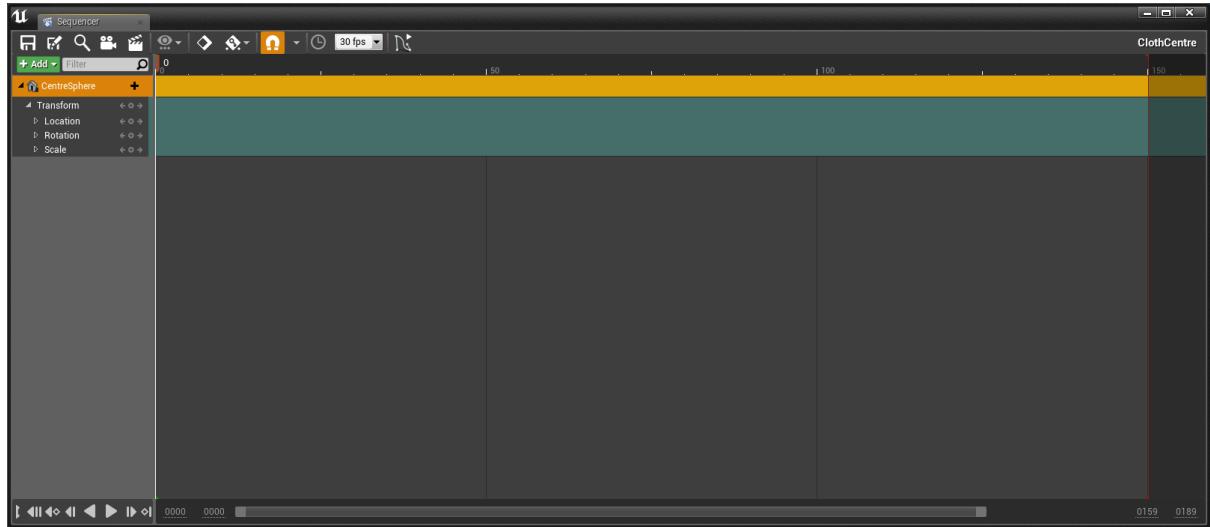


Figure 288

Added keys to track to move the sphere up and down to collide with cloth to demonstrate the dynamic collision.

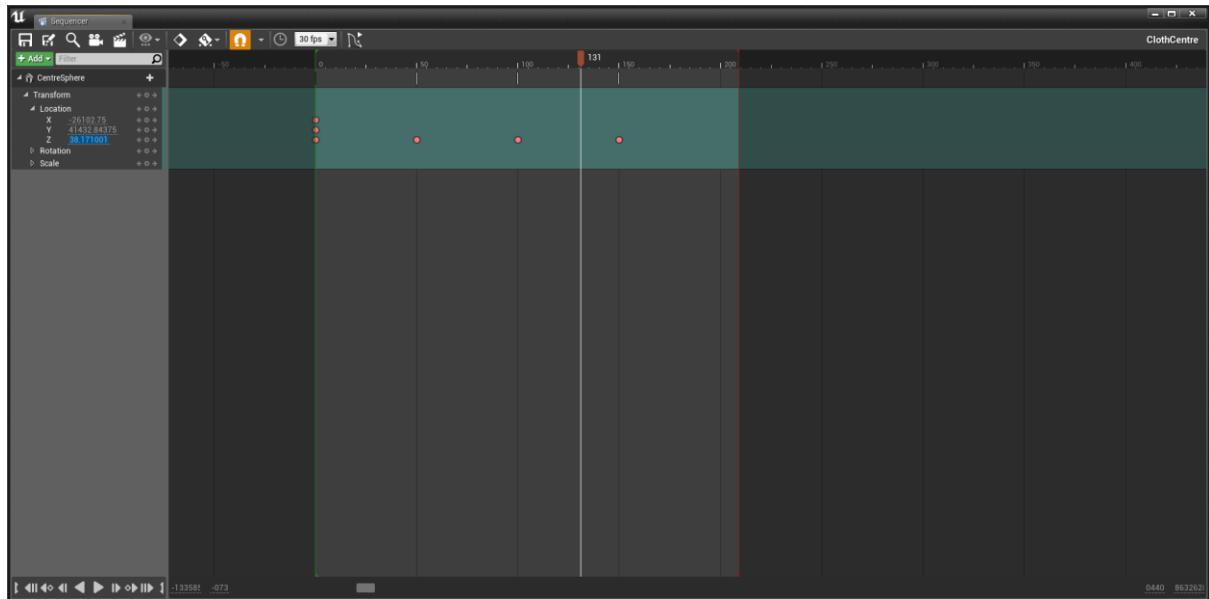


Figure 289

Playing the sequence and the sphere colliding with the cloth.

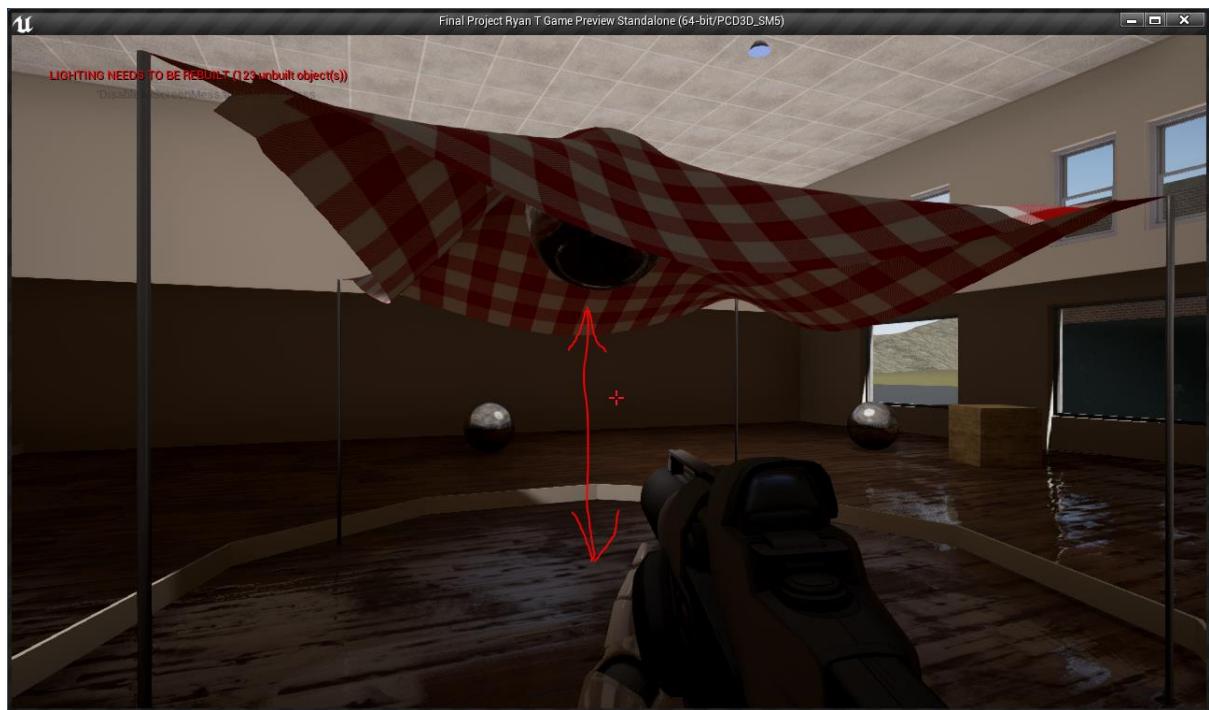


Figure 290

Added a second sphere for better visual effect.

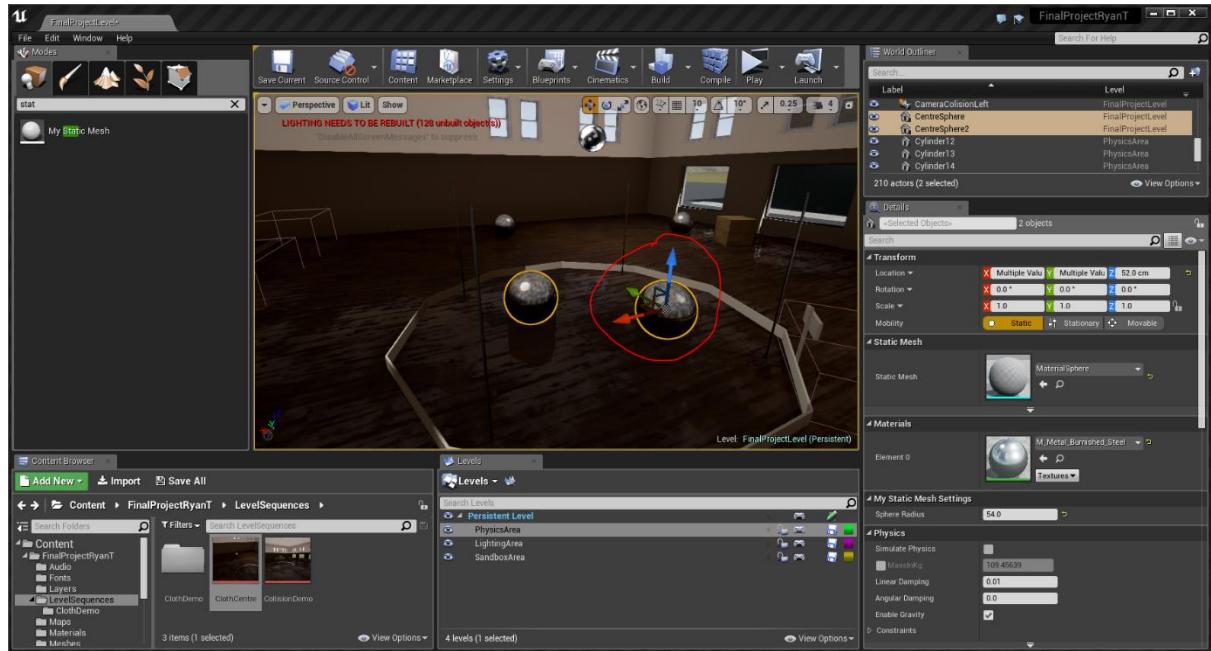


Figure 291

Added keys for 2nd sphere to move it up and down.

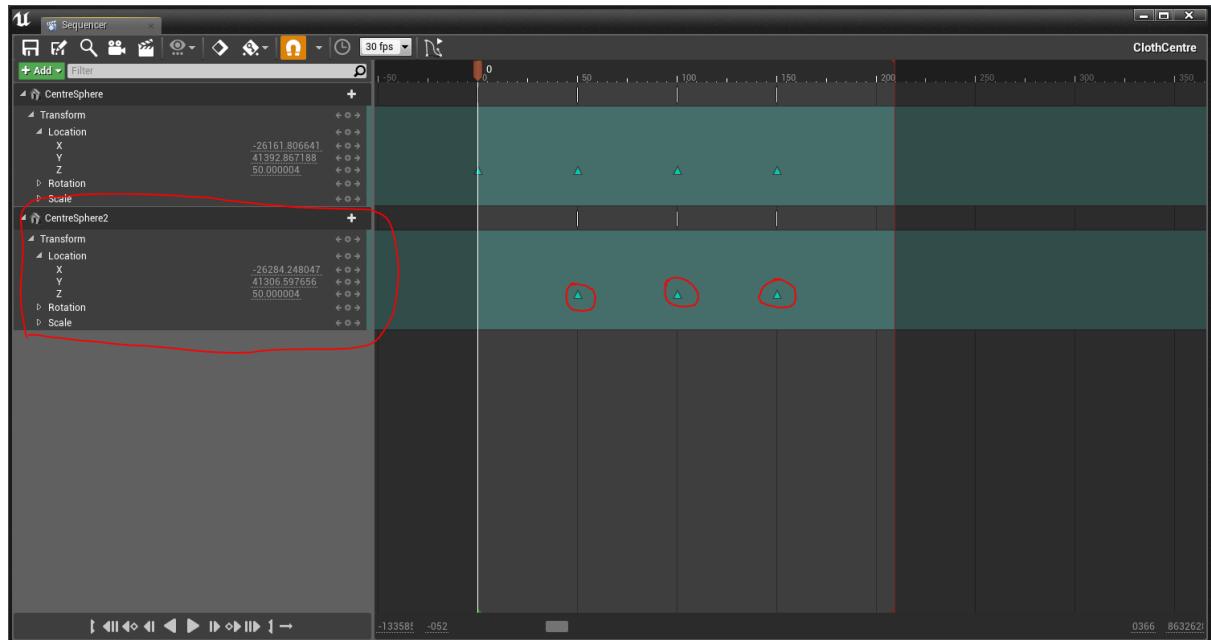


Figure 292

Playing the sequence and both sphere colliding with the cloth.



Figure 293

Added the centre cloth object to the sequence with a render mode track. The cloth now alternates between mesh and wireframe during sequence.

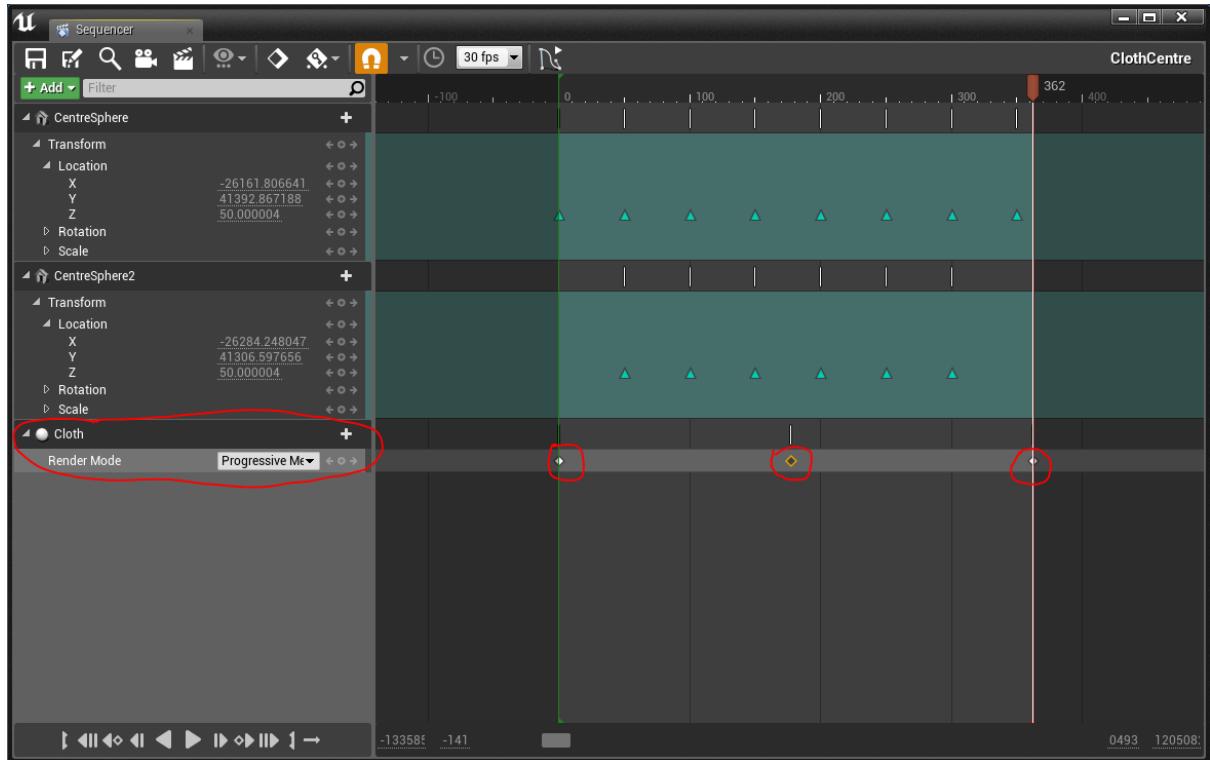


Figure 294

The cloth now in world in wireframe.

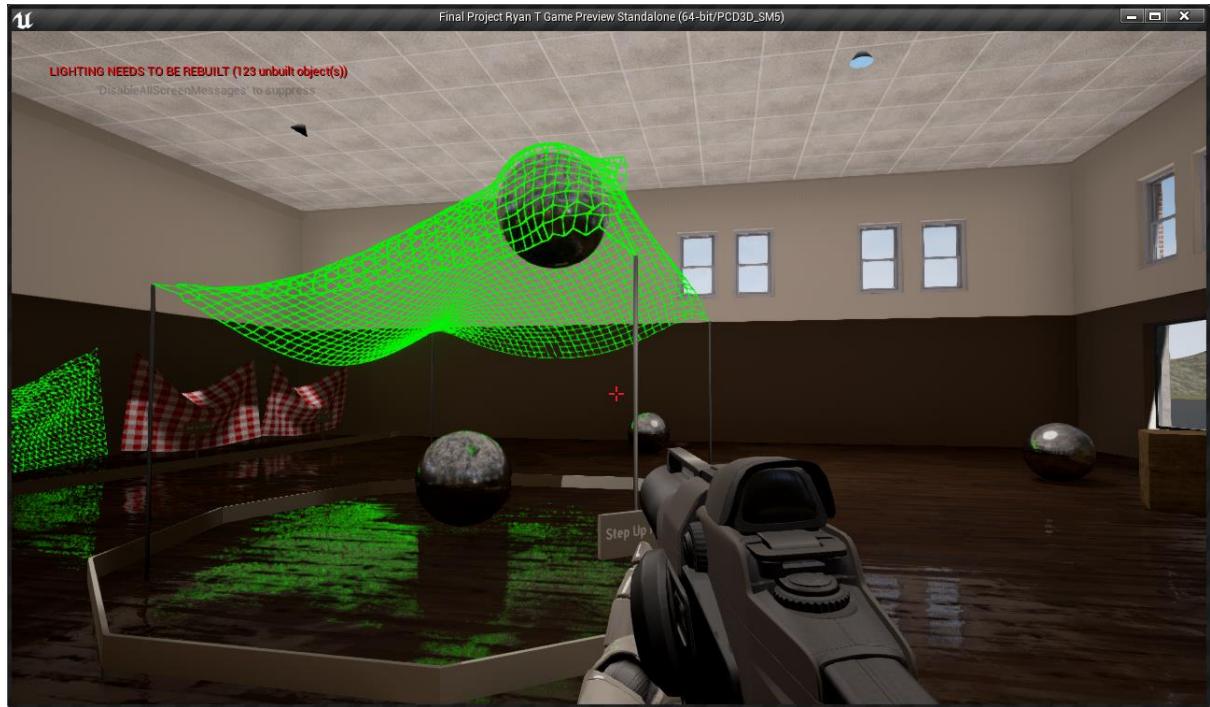


Figure 295

Added UTextContainer pointer to VerletSim class.

```
190     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
191         UMaterial* MeshMaterial = nullptr;
192
193     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
194         FVector2D UVTileScale = FVector2D(1.0f, 1.0f);
195
196     TArray<VerletConstraint> ConstraintList;
197
198     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
199         TArray<FVerletAttachment> AttachmentList;
200
201     UPROPERTY()
202         UTextContainerComponent* TextContainer = nullptr;
203
204 protected:
205     void SimulateVert(Verlet& Vert, float DeltaTime);
206     bool ResolveVertCollision(Verlet& Vert);
207     void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
208     void DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode );
209
210 };
211
```

Figure 296

Added code to VerletSim class in BeginPlay to find any attached UTextContainerComponents in the actor.

```

21 // ****
22 // BeginPlay - Called when the game starts or when spawned
23 // ****
24 void AVerletSim::BeginPlay()
25 {
26     Super::BeginPlay();
27
28     // Finds the first wind source object in the level
29     if (!Wind)
30     {
31         // GetAllActorsOfClass Searches the world and returns a list of all objects of the type required.
32         // In this case we are looking for all AWindSim objects.
33         TArray<Actor*> WindList;
34         UGameplayStatics::GetAllActorsOfClass(GetWorld(), AWindSim::StaticClass(), WindList);
35
36         if (WindList.Num())
37             Wind = Cast<AWindSim>(WindList[0]);    // Use the first one in the list
38     }
39
40     MeshComponent = NewObject<UProceduralMeshComponent>(this, NAME_None);
41     MeshComponent->RegisterComponent();
42     MeshComponent->SetWorldTransform(FTransform());    // Set transform to Position(0,0,0) and Rotation(0,0,0)
43
44     // KeepWorldTransform means that the component is not relative to the AVerletSim actor, it has its own worldspace position
45     MeshComponent->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);
46     LastAddedComponent = MeshComponent;
47
48     // Get a list of all UTextContainerComponents
49     TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
50
51     if (TextComponents.Num() )
52         TextContainer = Cast<UTextContainerComponent>(TextComponents[0]);
53 }

```

Figure 297

Added TextContainerIndex Var and UpdateSubTitleText function to VerletSim class (Header).

```

179     int RenderModeIndex = 0;
180
181     float Timer = 0.0f;           // General timer for animation debug stuff
182
183     AWindSim* Wind = nullptr;
184
185     UPrimitiveComponent* LastAddedComponent = nullptr;
186
187     UPROPERTY()
188     UProceduralMeshComponent* MeshComponent = nullptr;
189
190     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
191         UMaterial* MeshMaterial = nullptr;
192
193     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
194         FVector2D UVTileScale = FVector2D(1.0f, 1.0f);
195
196     TArray<VerletConstraint> ConstraintList;
197
198     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
199         TArray<FVerletAttachment> AttachmentList;
200
201     UPROPERTY()
202         UTextContainerComponent* TextContainer = nullptr;
203
204     UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Text")
205         float TextContainerIndex = -1;                                // -1 default = No Text
206
207 protected:
208     void SimulateVert(Verlet& Vert, float DeltaTime);
209     bool ResolveVertCollision(Verlet& Vert);
210     void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
211     void DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode );
212     void UpdateSubTitleText();
213
214 };

```

Figure 298

Added UpdateSubTitleText function to VerletSim class (Called from Tick) (CPP file)

```
56 // ****
57 // Tick - Called every frame
58 // ****
59 void AVerletSim::Tick(float DeltaTime)
60 {
61     Super::Tick(DeltaTime);
62
63     Timer += DeltaTime;
64
65     if ( RenderMode == eRenderMode::ProgressiveMesh )
66         MeshComponent->SetVisibility( true, true );
67     else
68         MeshComponent->SetVisibility( false, true );
69
70     if ( bRenderModeIncremental )
71     {
72         RenderModeTimer += DeltaTime;
73         if ( RenderModeTimer >= RenderModeIncrementalTime )
74         {
75             RenderModeTimer -= RenderModeIncrementalTime;
76             RenderModeIndex++;
77
78             UE_LOG(LogTemp, Warning, TEXT("%d"), RenderModeIndex );
79
80         }
81     }
82
83     MeshComponent->SetWorldTransform(FTransform());      // Set transform to Position(0,0,0) and Rotation(0,0,0)
84
85     UpdateSubTitleText();
86 }
87
88 // ****
89 // UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
90 // ****
91 void AVerletSim::UpdateSubTitleText()
92 {
93     if (TextContainer)
94     {
95         TextContainer->SetTextFromList(-1);           // Turn off 3d text
96
97         FString Temp = TextContainer->GetTextFromList(TextContainerIndex);
98
99         // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
100        while ( true )
101        {
102            int32 Index;
103            if ( !Temp.FindChar( TCHAR('\r'), Index ) )
104                break;
105            Temp.RemoveAt( Index, 1 );
106        }
107        AFinalProjectRyanTHUD::SetString( Temp );
108    }
109 }
```

Figure 299

Added a Text Container component to the centre cloth actor and added 4 placeholder texts.



Figure 300

Added a new track to centre cloths level sequence for text container index. This allows the text to change index during the sequence.

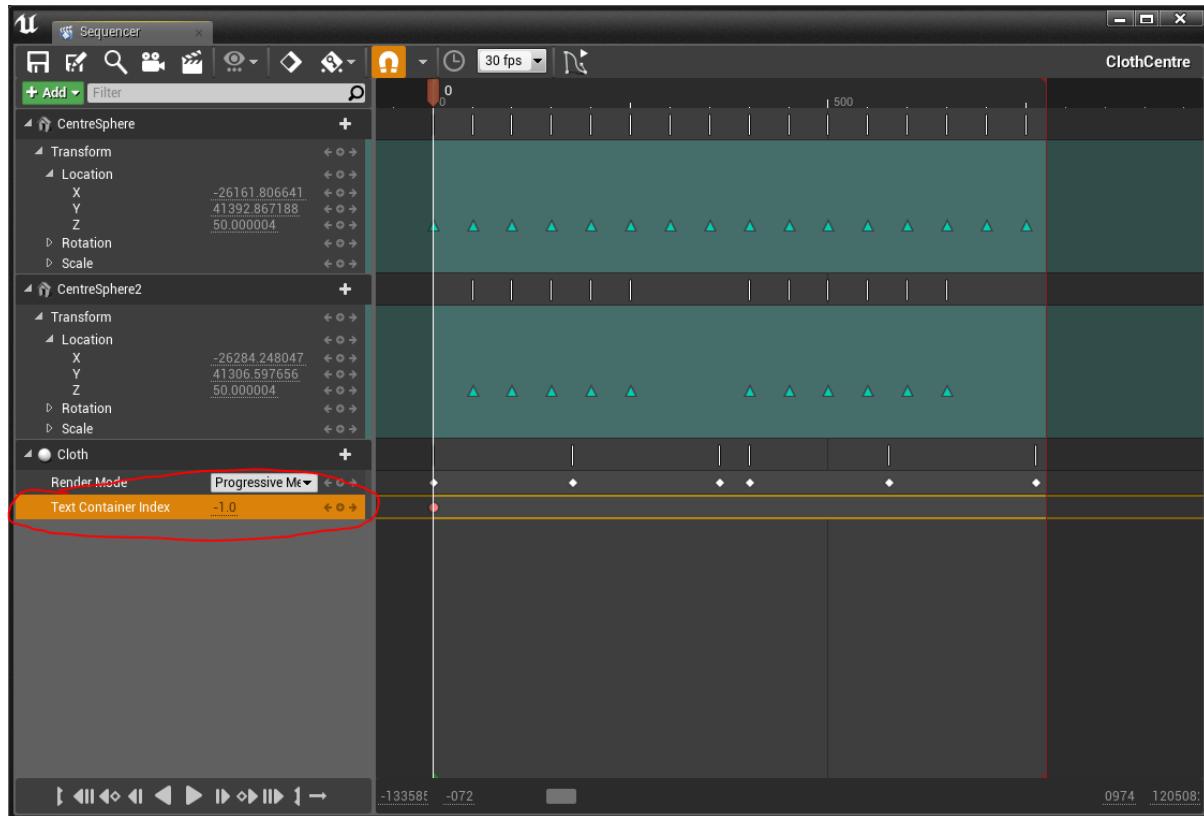


Figure 301

Added keys to the track to cycle -1 0 1 2 3 -1 through the placeholder text.

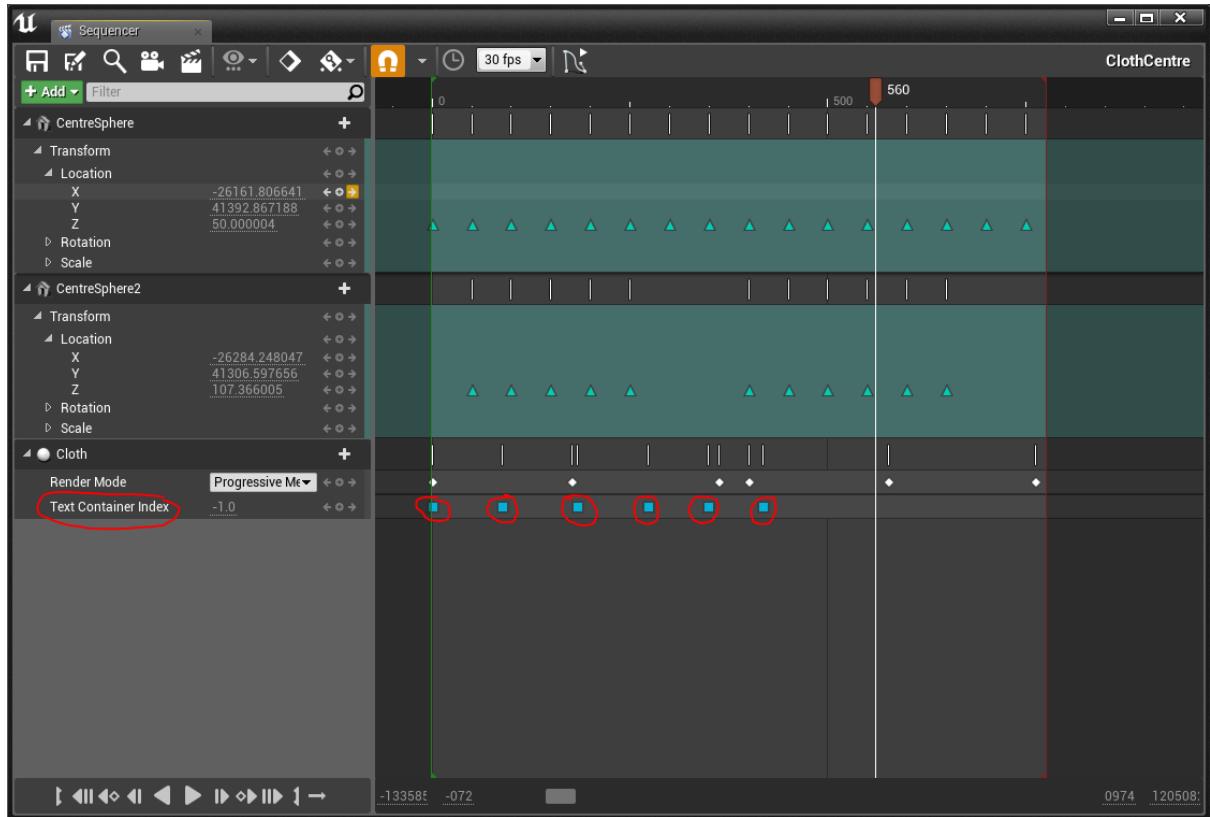


Figure 302

Sub Title text in place during the sequence.

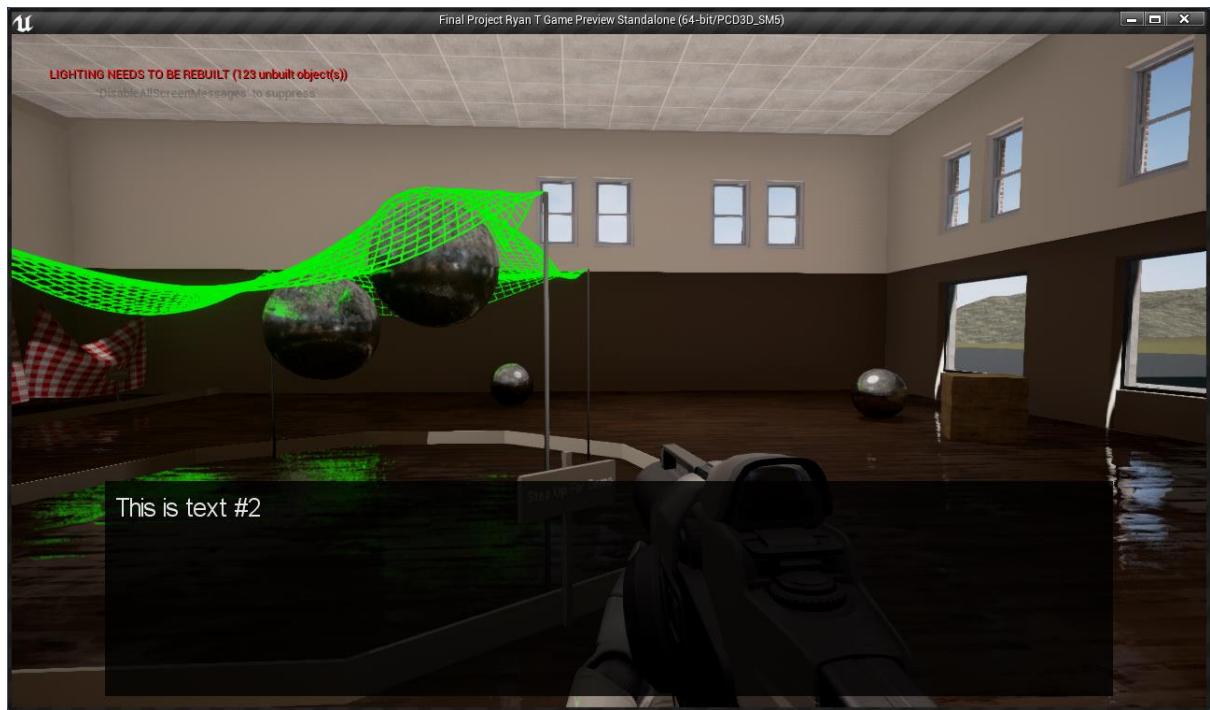


Figure 303

Added the final text and update level sequence keys to get timing correct.

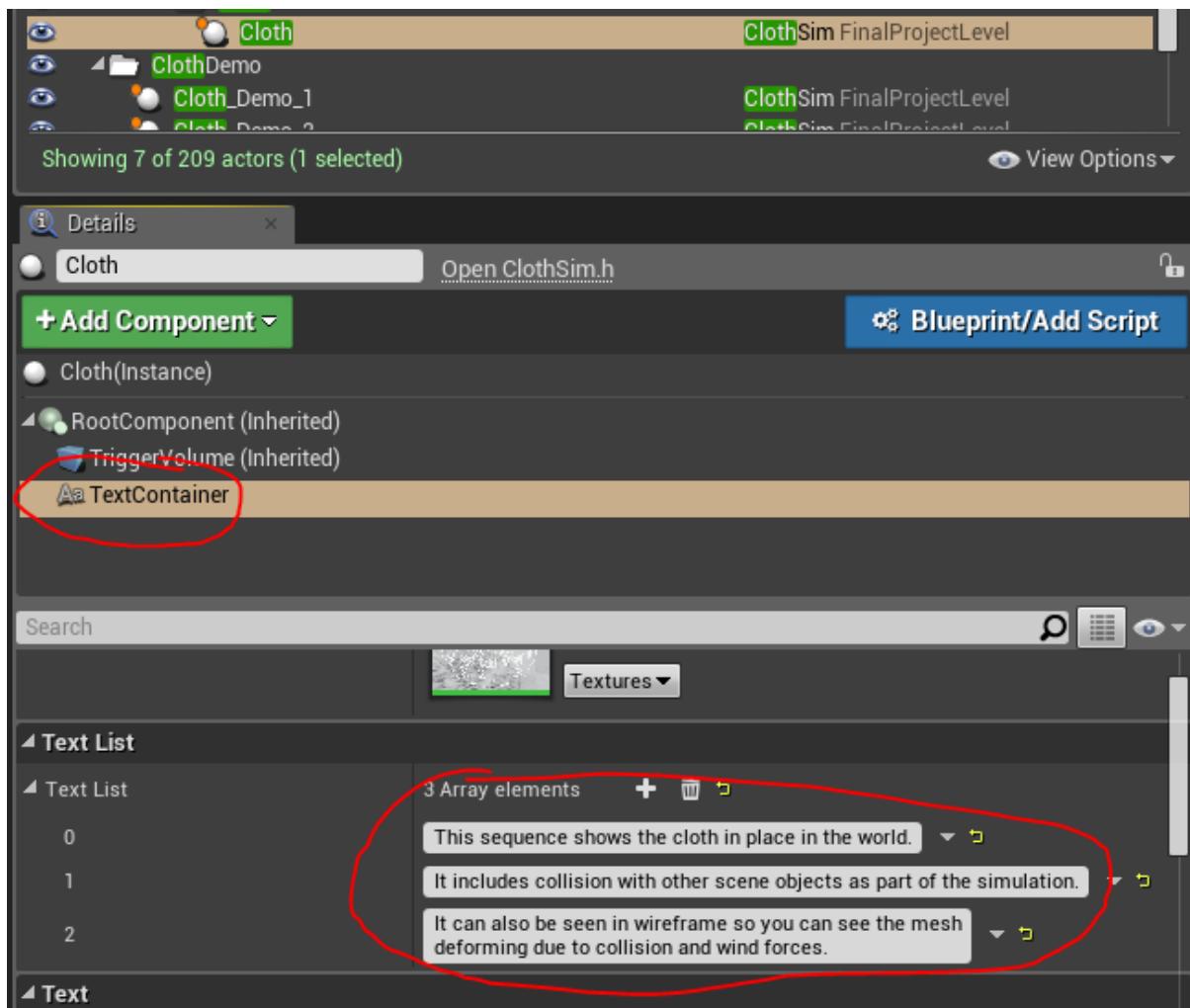


Figure 304

Created 6 Cloth Actors, All with TextContainer components attached and level sequence files setup ClothDemo\_1,2,3,4,5,6. Now ready for sequencing.

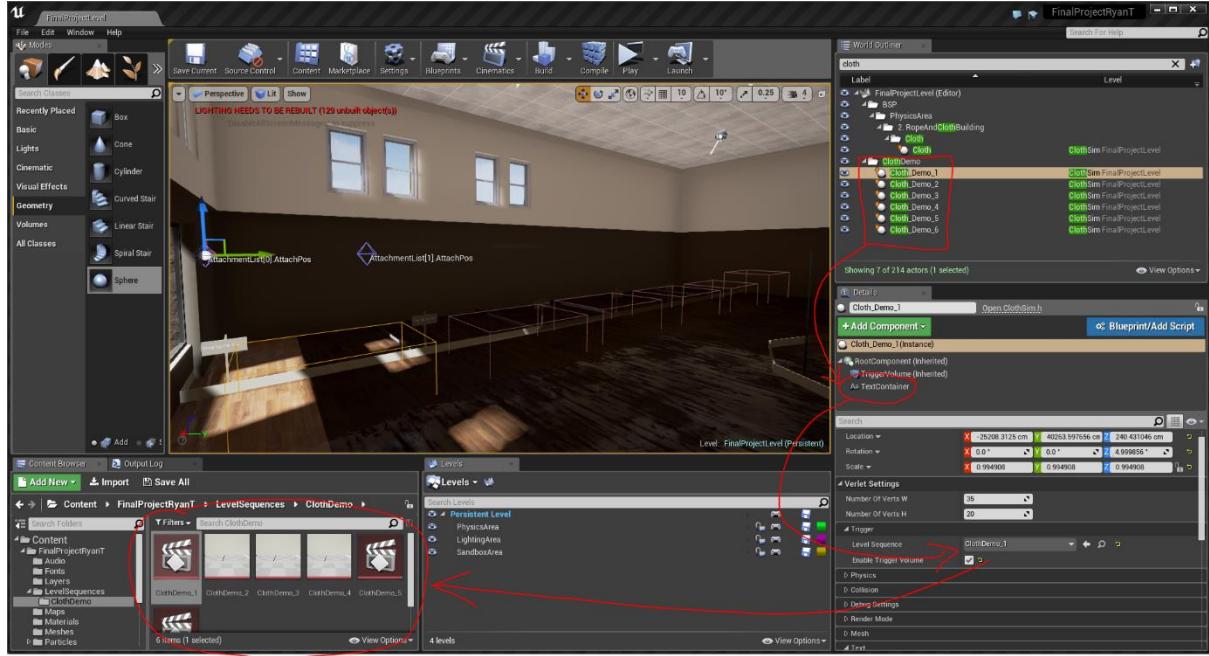


Figure 305

Started sequence 1 by adding some text to the cloth text container ready for sequencing.

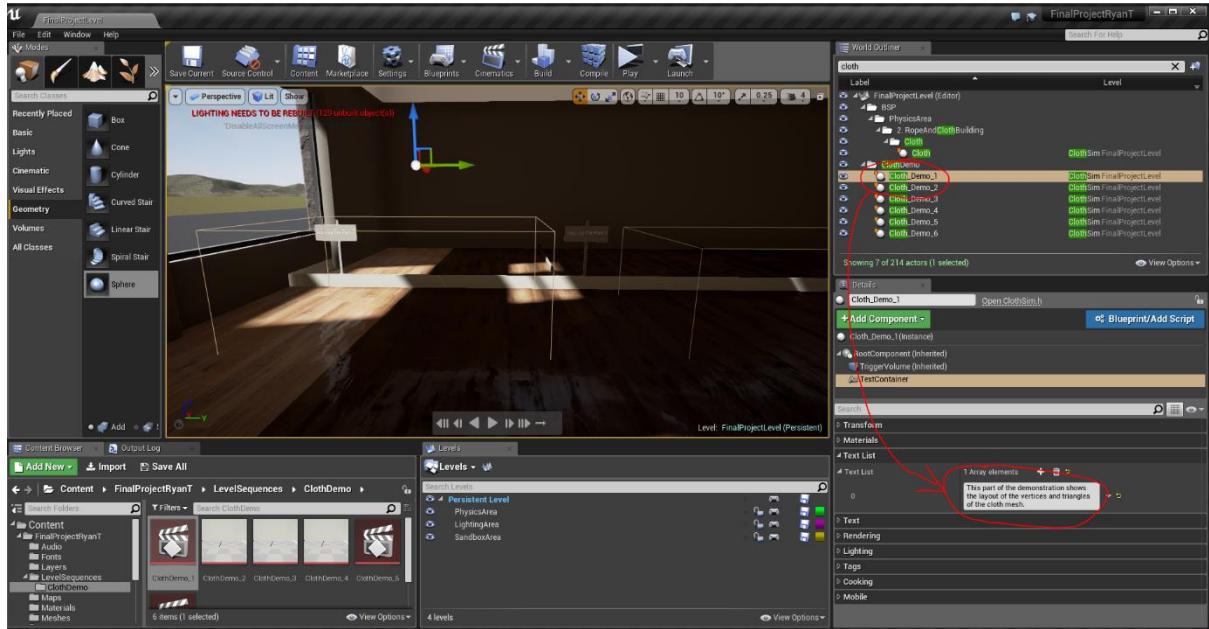


Figure 306

Added Cloth\_Demo\_1 actor to the level sequence and added a TextContainerIndex track. Added Key to switch to text index 0 sometime after start.

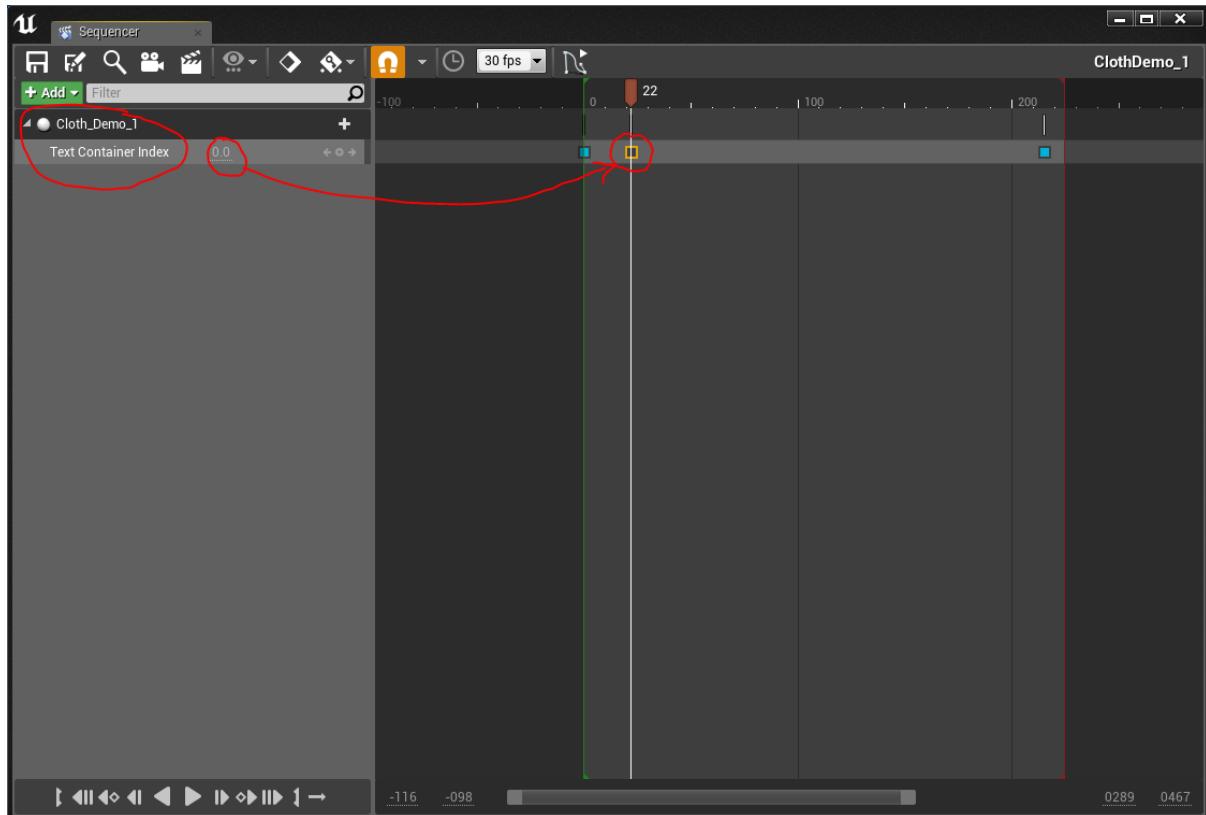


Figure 307

Shot of the text being displayed in HUD, triggered from level sequence.

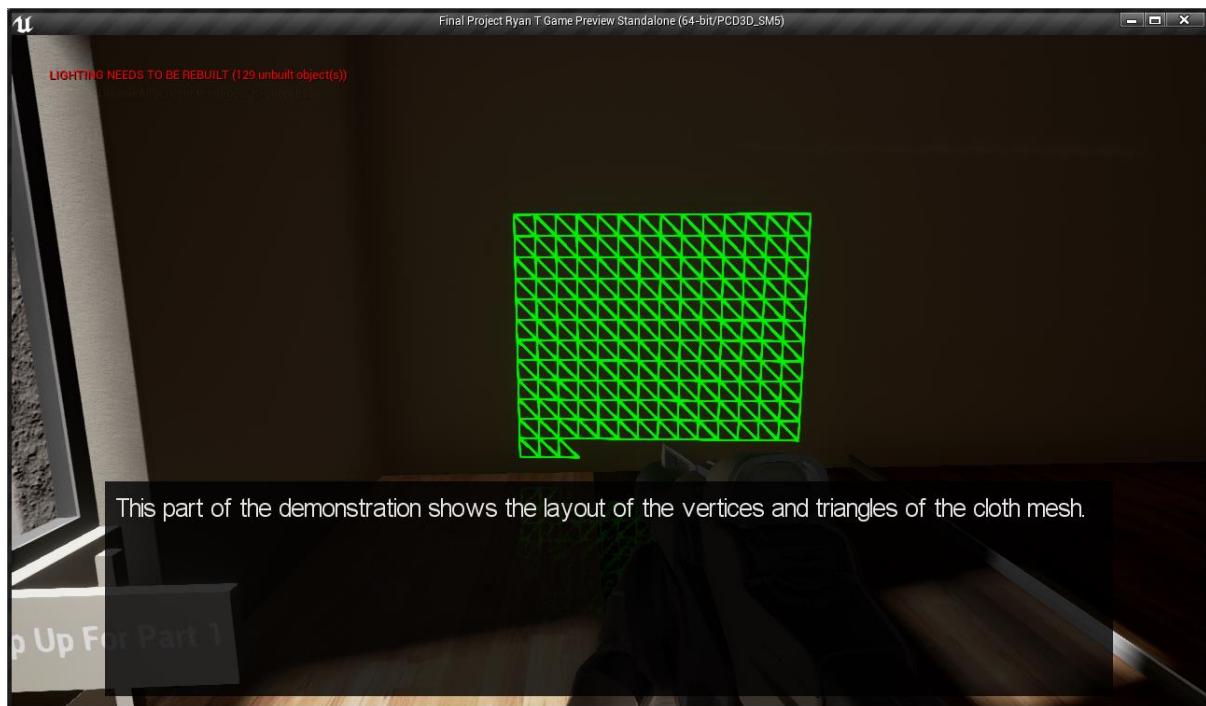


Figure 308

Added a second piece of text and added it to the level sequence track.

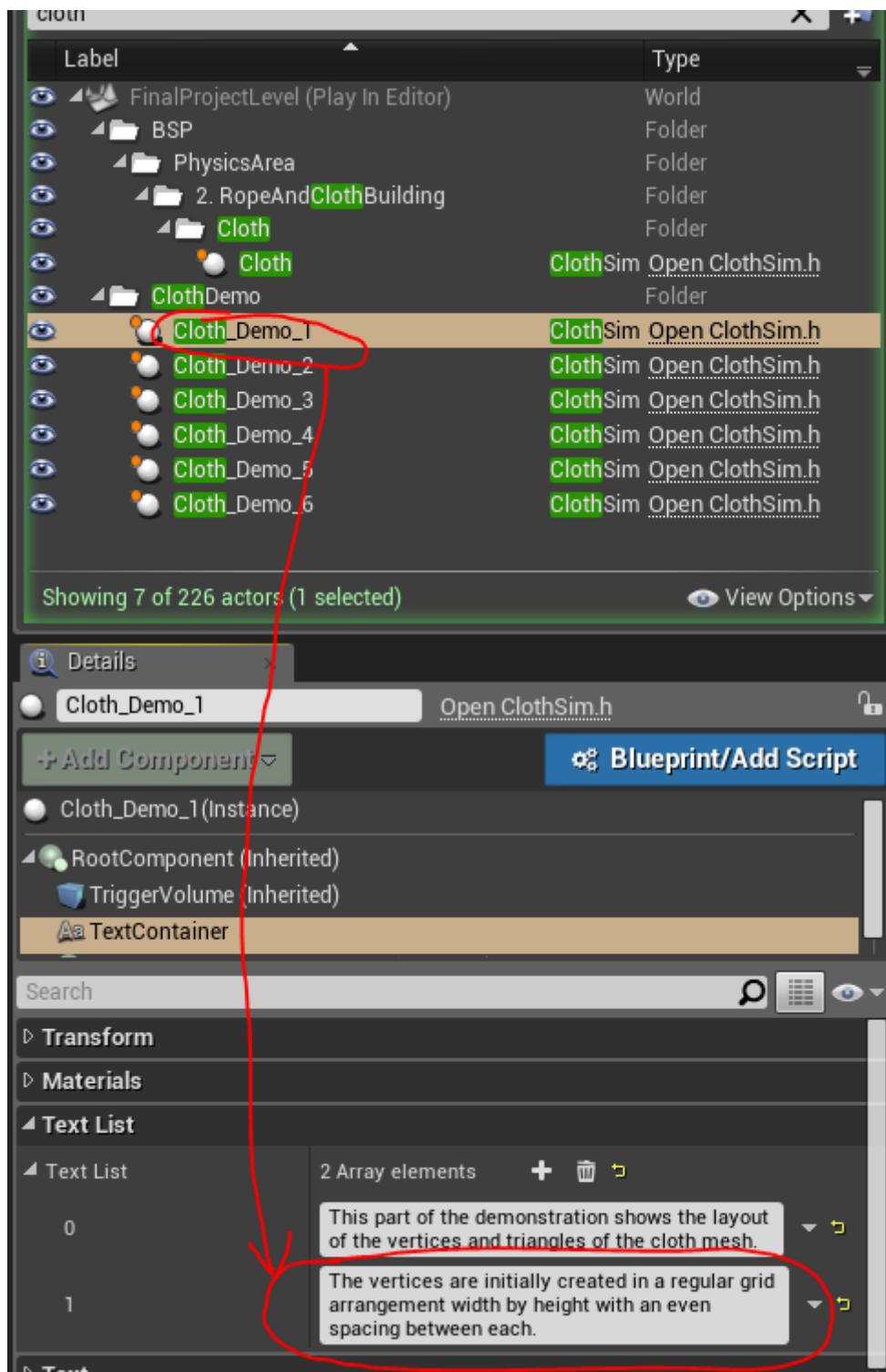


Figure 309

Second text in place.



Figure 310

Added relevant text to the 2nd cloth actor ready for sequencing.

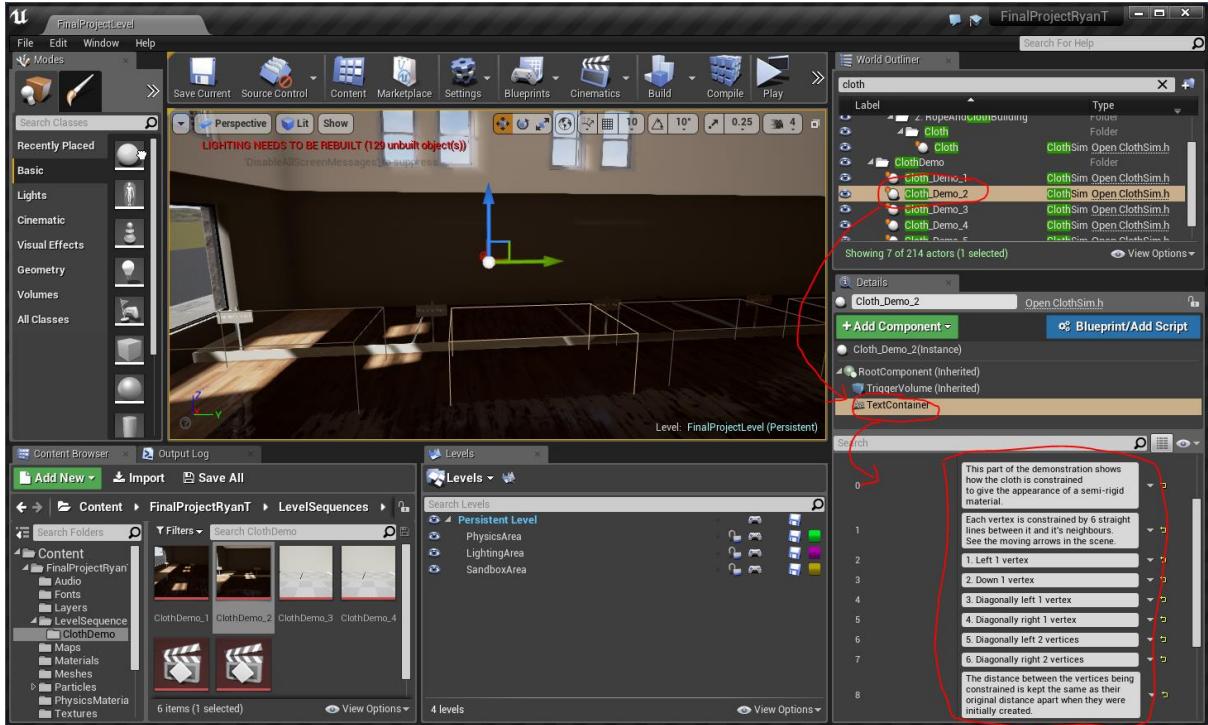


Figure 311

Added Draw Constraint Filter, RenderMode, TextContainerIndex tracks to demo sequence 2. Added keys to show text and display the different constraints.

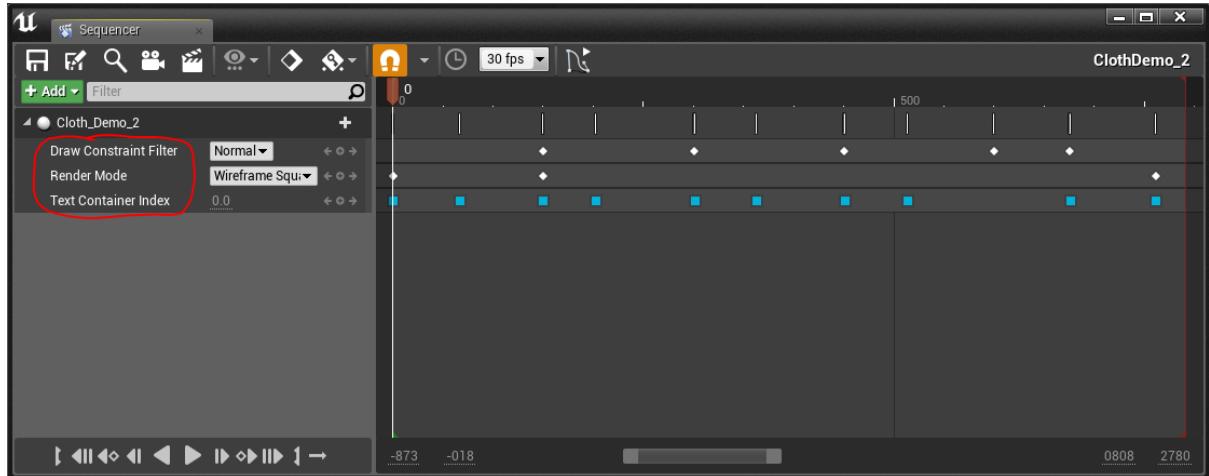


Figure 312

Added code to turn off HUD subtitles when sequence finished or trigger volume was exited.

```
88 // ****
89 // OnEndSequenceEvent - Triggered when level sequence has finished playing
90 // ****
91 void AClothSim::OnEndSequenceEvent()
92 {
93     SequencePlayer->Stop();
94     SequencePlayer = nullptr;
95
96     TextContainerIndex = -1;
97     AFinalProjectRyanTHUD::SetString();
98 }
99
100 // ****
101 // EnterVolume - Called when component exits collision volume
102 // ****
103 void AClothSim::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
104 {
105     if ( SequencePlayer )
106     {
107         SequencePlayer->Stop();
108         SequencePlayer = nullptr;
109     }
110     TextContainerIndex = -1;
111     AFinalProjectRyanTHUD::SetString();
112 }
```

Figure 313

Added Cameras for first 2 cloth stages.

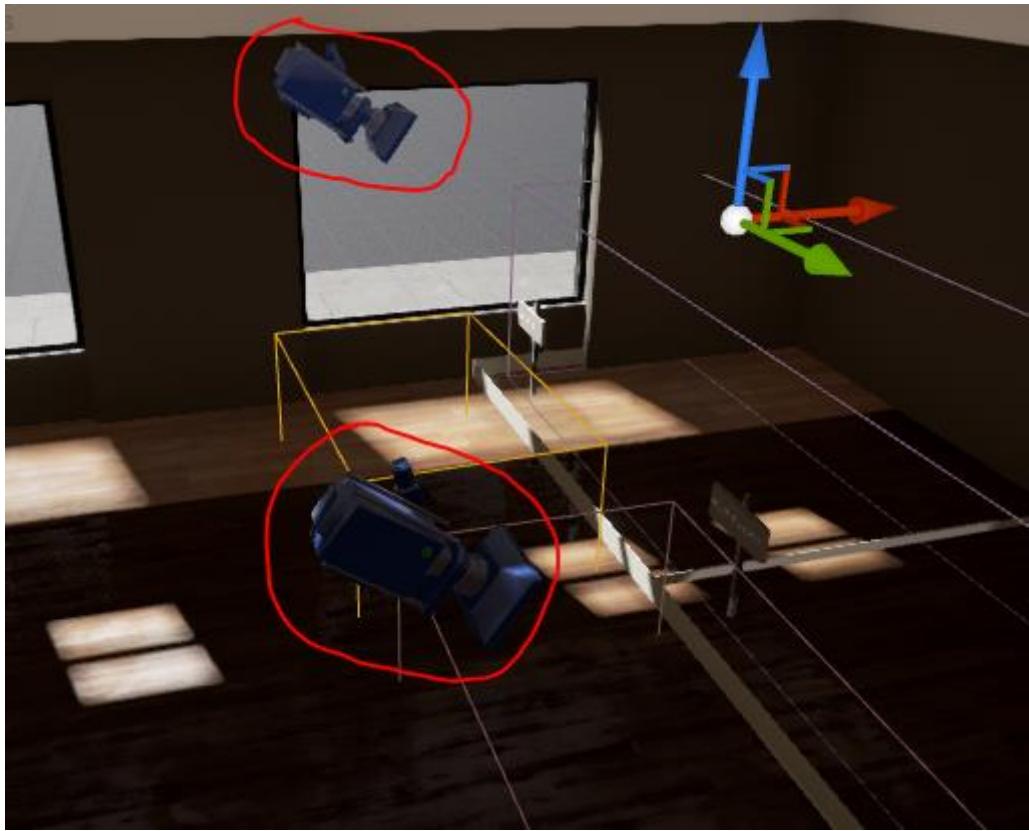


Figure 314

Added Cameras the cameras to their relevant level sequence in a camera cut track so that the camera cuts to focus on demo when it starts.

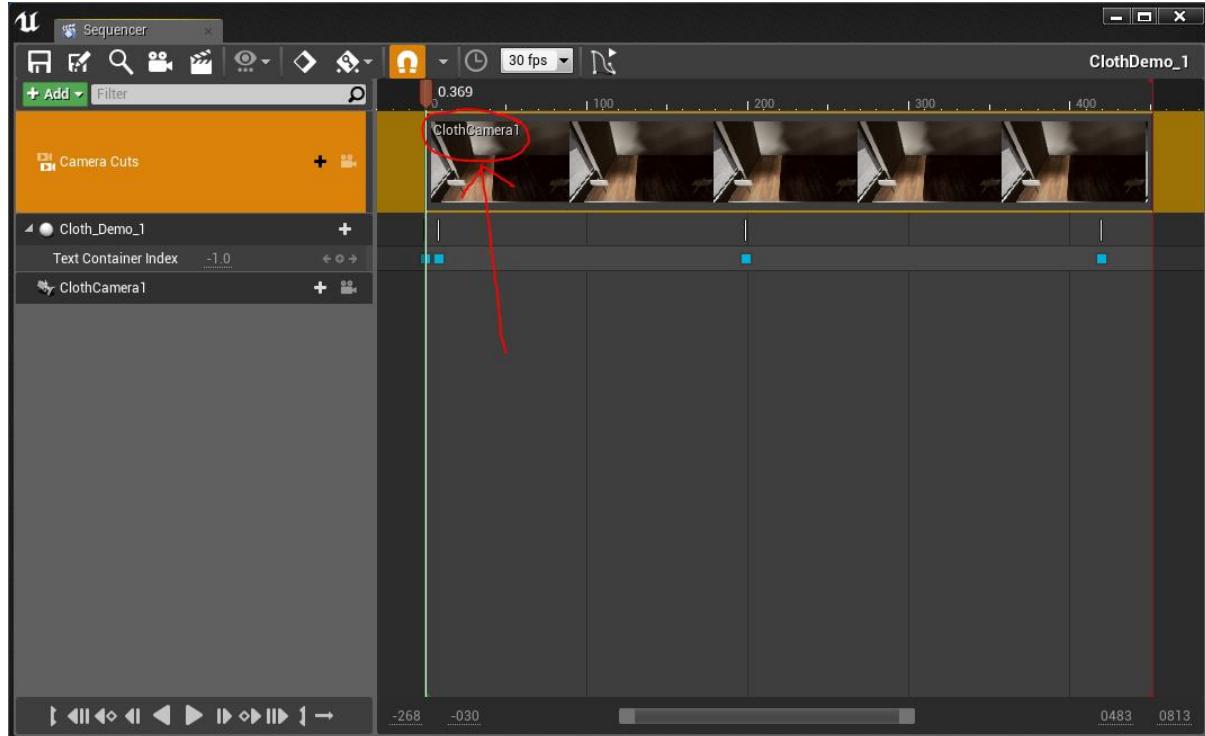


Figure 315

Showing the camera cut with demo stage 1.

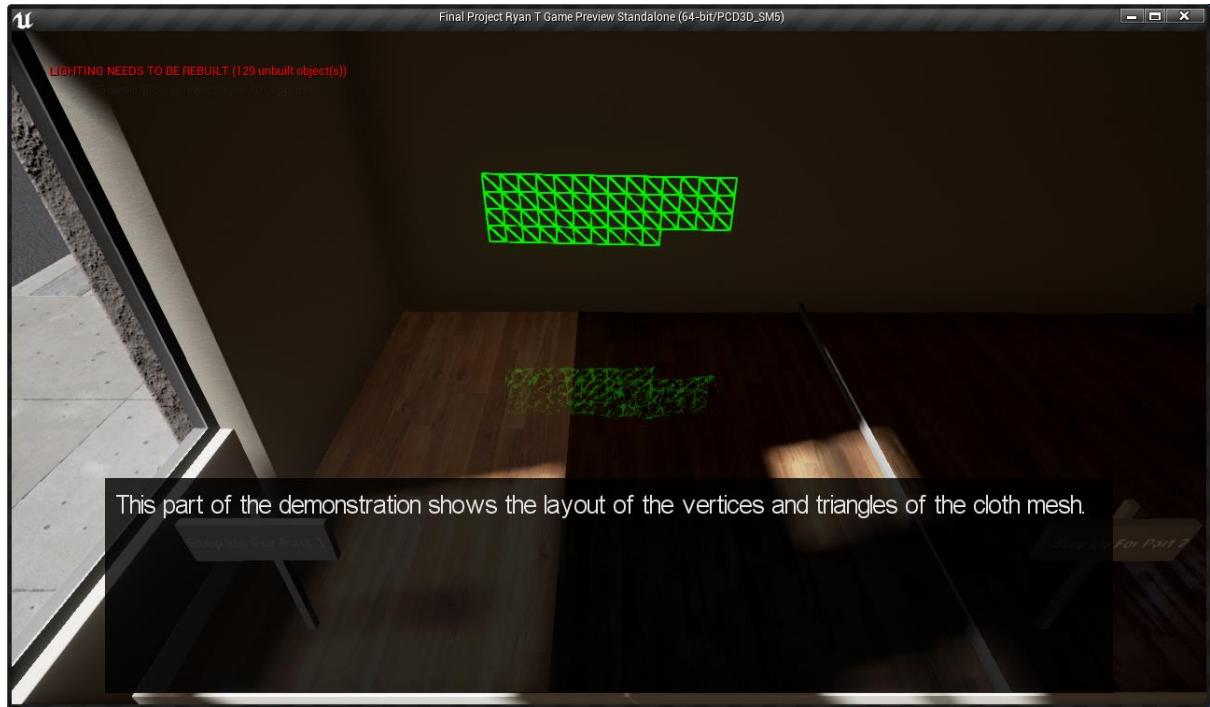


Figure 316

BUG FIX. Added checks in various functions to stop actors other than characters from triggering level sequence.

```
72 //*****
73 // EnterVolume - Called when component enters collision volume
74 //*****
75 void AClothSim::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
76 {
77     // Only allowed if overlapped by any kind of character
78     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
79         return;
80
81     if ( LevelSequence )
82     {
83         FLevelSequencePlaybackSettings Settings;
84
85         Settings.LoopCount = 0;
86
87         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
88         SequencePlayer->Play();
89     }
90 }
```

Figure 317

Added cut cameras to the other stages.



Figure 318

Changed VerletSim DrawForce to take a colour param.

```
287 // ****
288 // DrawForce - Draws a debug arrow to show force in world
289 // ****
290 void AVerletSim::DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode, FColor Colour )
291 {
292     if ( Mode == eShowForceMode::None )
293         return;
294
295     float Magnitude = Force.Size();                                // Get the magnitude of the force (Speed)
296
297     if ( Magnitude < 0.01f )                                         // If its too small to draw, just exit
298         return;
299
300     FVector Dir = Force / Magnitude;                               // Normalise dir by dividing by its magnitude
301
302     float Length = 30.0f;                                           // Maximum length
303
304     if ( Mode == eShowForceMode::ShowWithMagnitude )
305     {
306         // Scale Length my force magnitude
307         Length = 10.0f * Magnitude;
308     }
309     else
310     {
311         if ( Mode == eShowForceMode::ShowWithModulation )
312         {
313             // Apply some modulation to the length if needed
314
315             float Modulation = sinf(Timer*2.0f);           // Get modulating value from -1 to +1
316             Modulation += 1.0f;                            // Add 1 to put in range 0 to 2
317             Modulation *= 0.5f;                           // divide by 2 to put in range 0 to 1
318             Length *= Modulation;                        // Scale Length by the modulating value from 0.0 to 1.0
319         }
320     }
321
322     Dir *= Length;                                              // Multiple Dir by Length to get offset to the end of the arrow
323
324     DrawDebugDirectionalArrow( GetWorld(), Pos, Pos + Dir, 7.0f, Colour, false, 0.0f, 0, 1.0f );
325
326
327 }
```

Figure 319

Added text to cloth demo object 3 to begin sequencing.

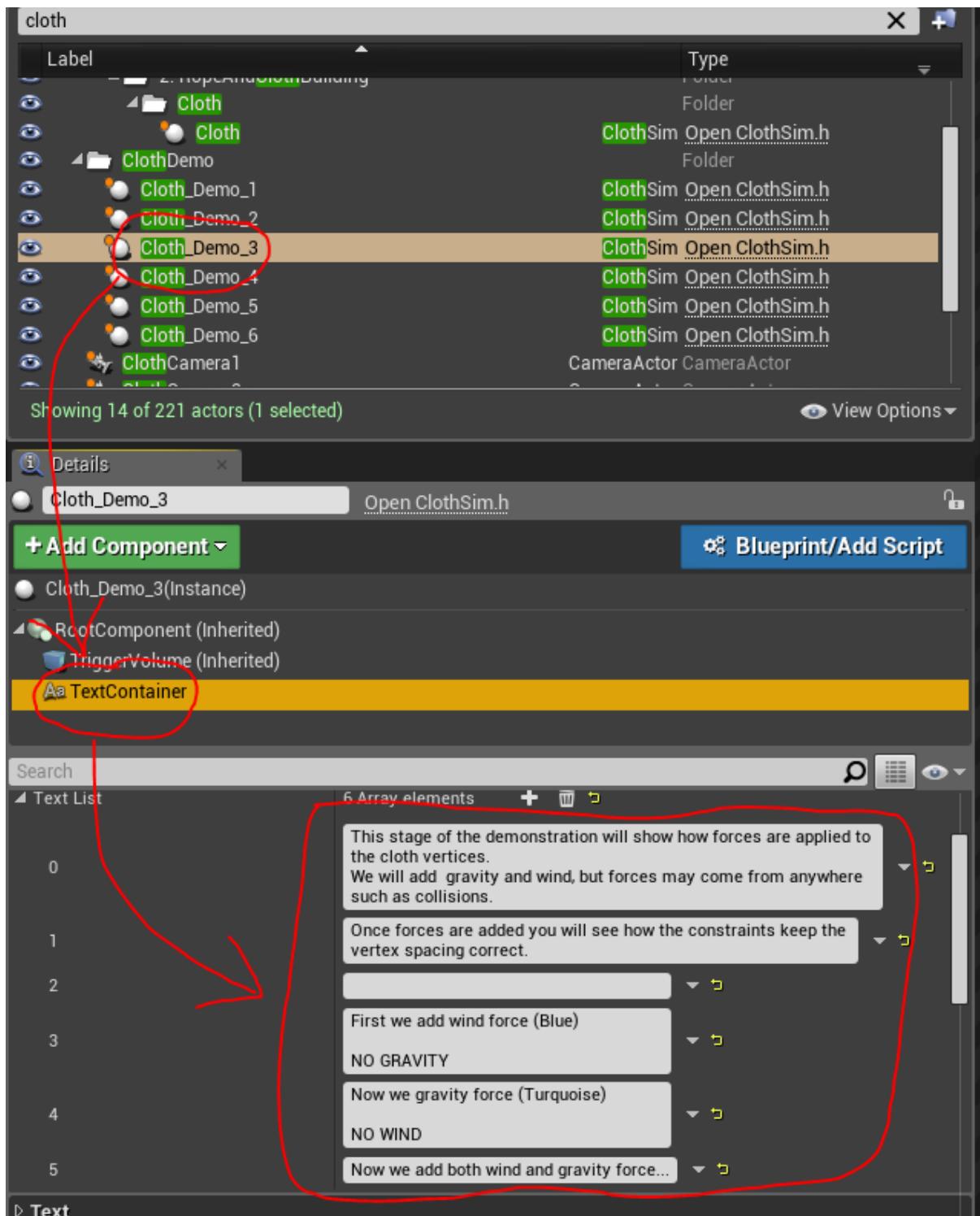


Figure 320

Setup sequence tracks cloth demo 3. Includes wind scale, enable gravity, text and show forces tracks. These are all exposed in code in the cloth code.

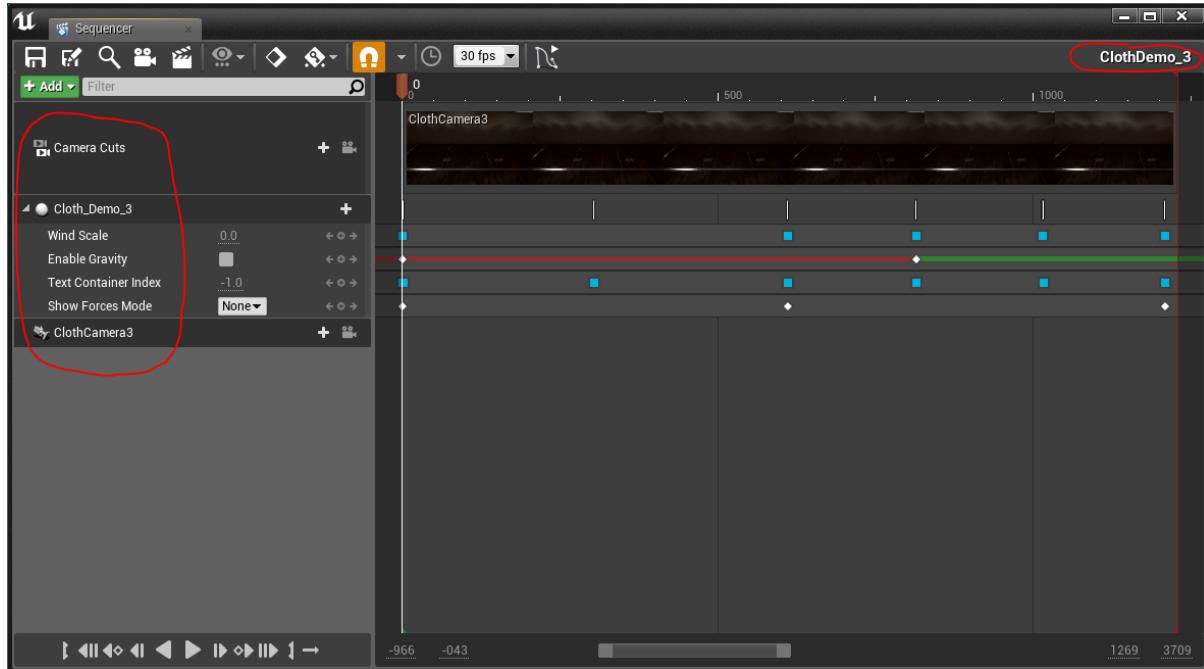


Figure 321

Various class variables exposed to editor timeline using the Unreal Engine UPROPERTY macro with the Interp keyword (Show in Timeline).

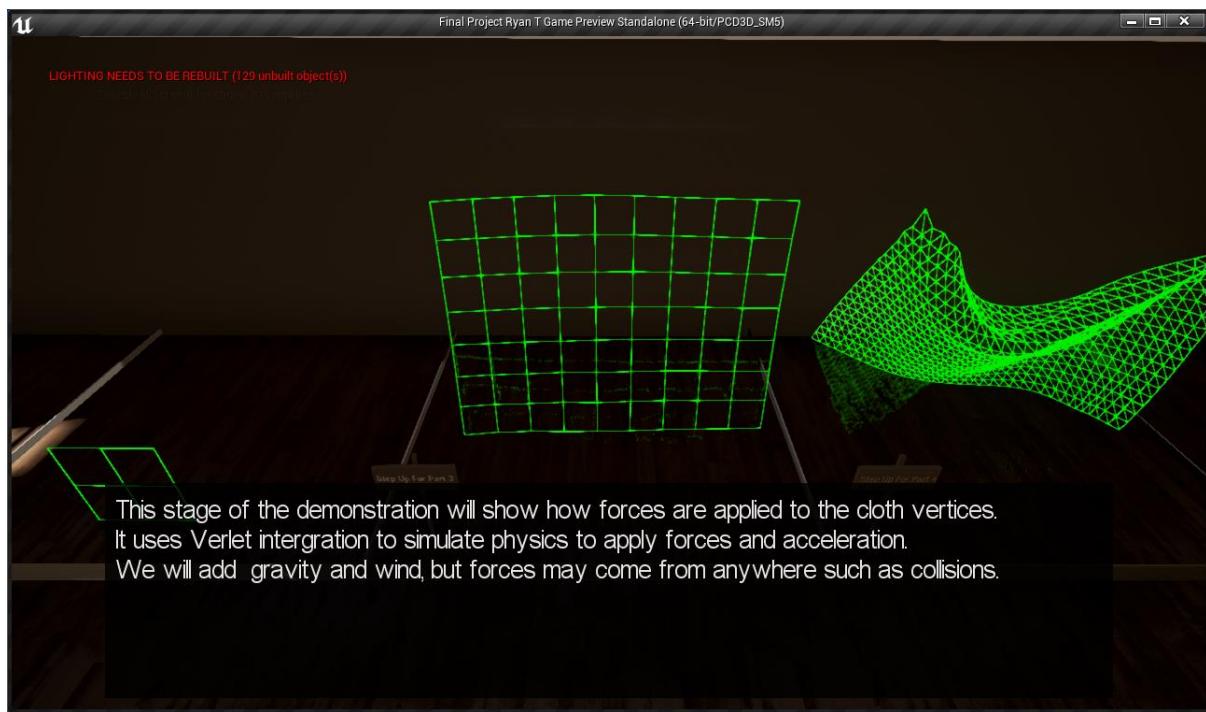
```

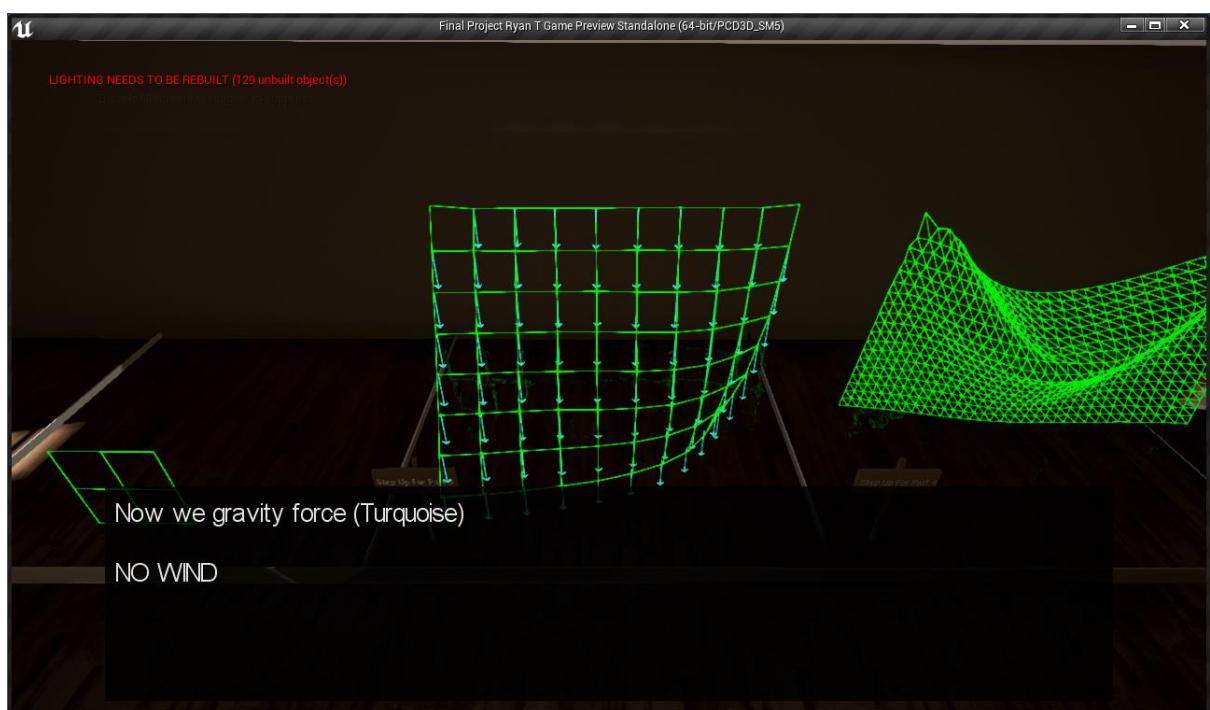
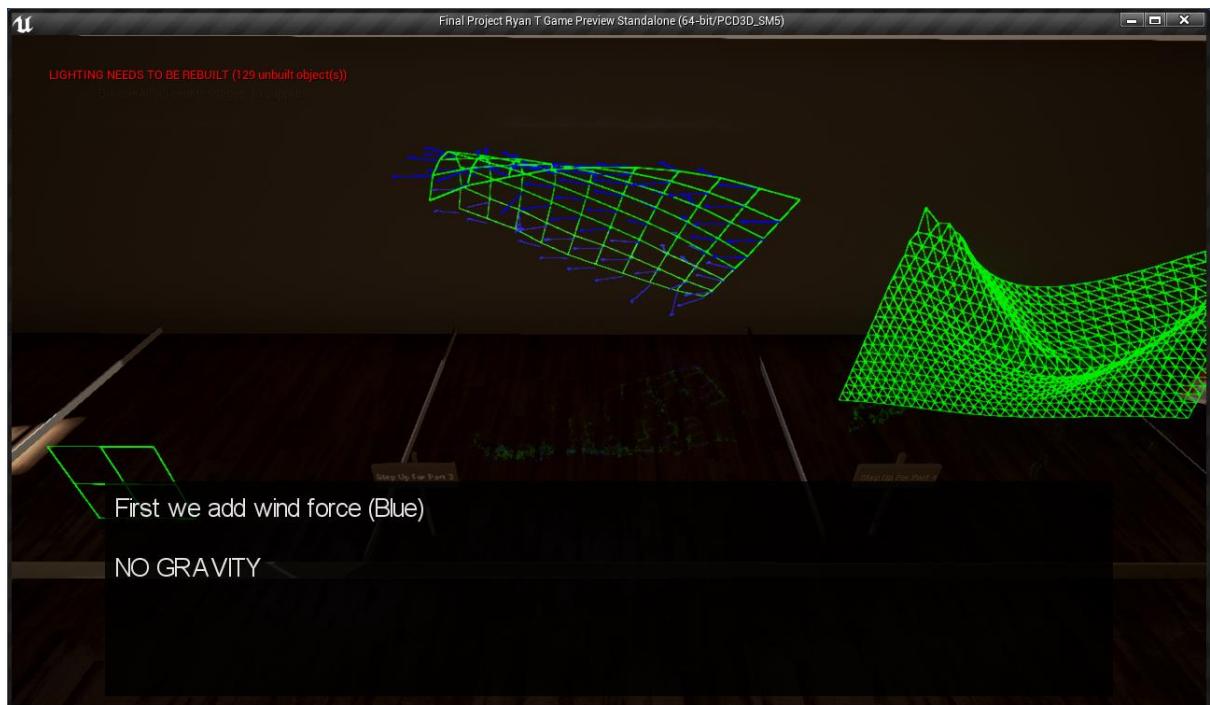
140
141     UPROPERTY(EditAnywhere,Interp, Category = "Verlet Settings|Physics")
142         float WindScale = 1.0f;
143
144     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
145         bool bEnableSimulation = true;
146
147     UPROPERTY(EditAnywhere,Interp , Category = "Verlet Settings|Physics")
148         bool bEnableGravity = true;
149
150 // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
151 // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.
152
153 // Toggles display of debug draw lines
154     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings", meta = (DisplayName = "Enable Debug Draw"))
155         bool bEnableDebugDraw = false;
156
157     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings")
158         bool bShowLockedVerts = false;
159
160     UPROPERTY(EditAnywhere,Interp , Category = "Verlet Settings|Debug Settings")
161         eShowForceMode ShowForcesMode = eShowForceMode::None;
162
163     UPROPERTY(EditAnywhere,Interp , Category = "Verlet Settings|Debug Settings")
164         eVerletConstraintType DrawConstraintFilter = eVerletConstraintType::All;
165
166     UPROPERTY(EditAnywhere,Interp , Category = "Verlet Settings|Render Mode")
167         eRenderMode RenderMode = eRenderMode::ProgressiveMesh;
168

```

Figure 322

Five screenshots showing the finished sequence.





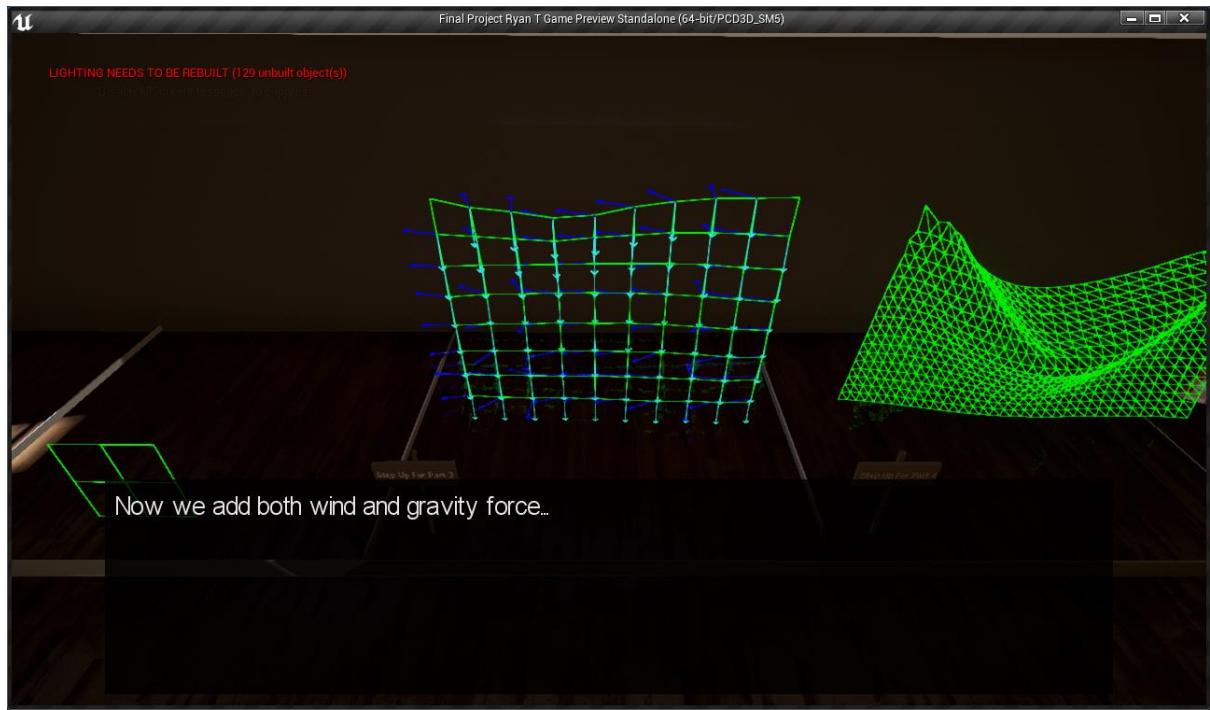


Figure 323 – Completed Cloth Demonstration (5 Images)

Verlet Intergration code. As seen in previous code.

```
Vert.Pos += ((Vert.Pos - Vert.PreviousPos) * DampingAmt) + (Force * DeltaTime);
```

Figure 324

Added text to cloth demo object 4 to begin sequencing.

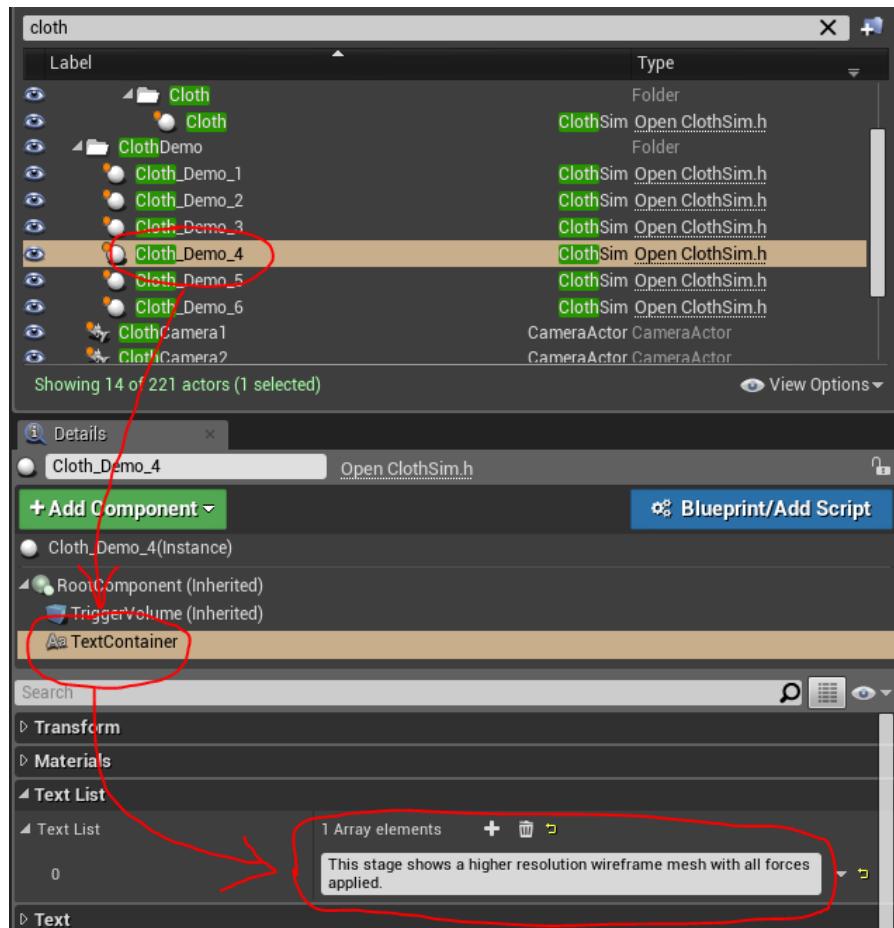


Figure 325

Setup sequence tracks cloth demo 4. Simply cuts the camera, displays the text and switched the cloth render mode from off to wireframe squares.

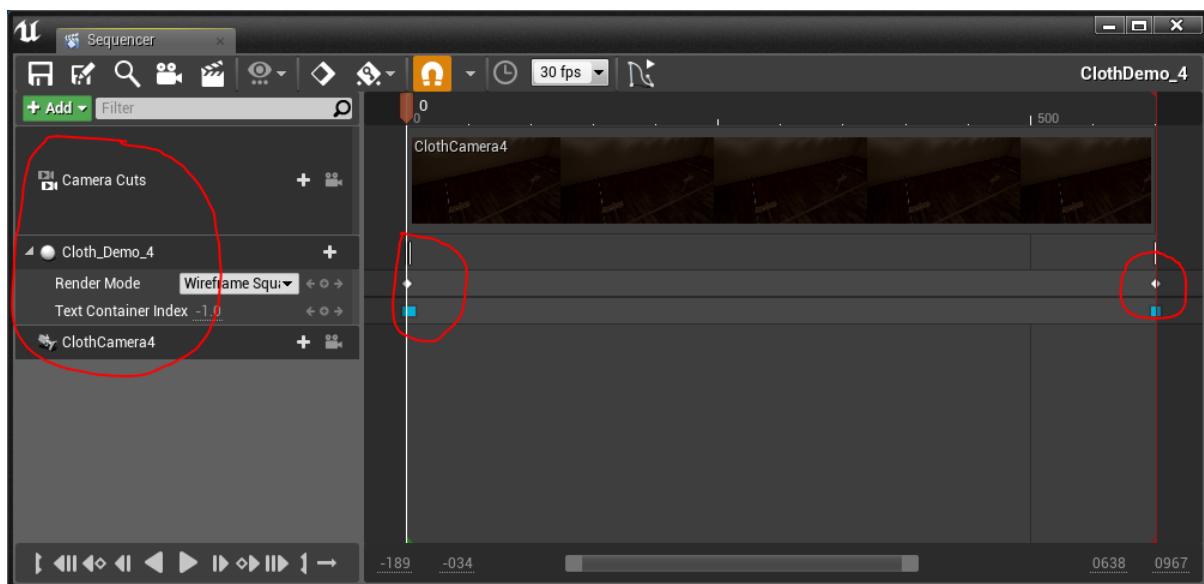


Figure 326

Showing Sequence 4.

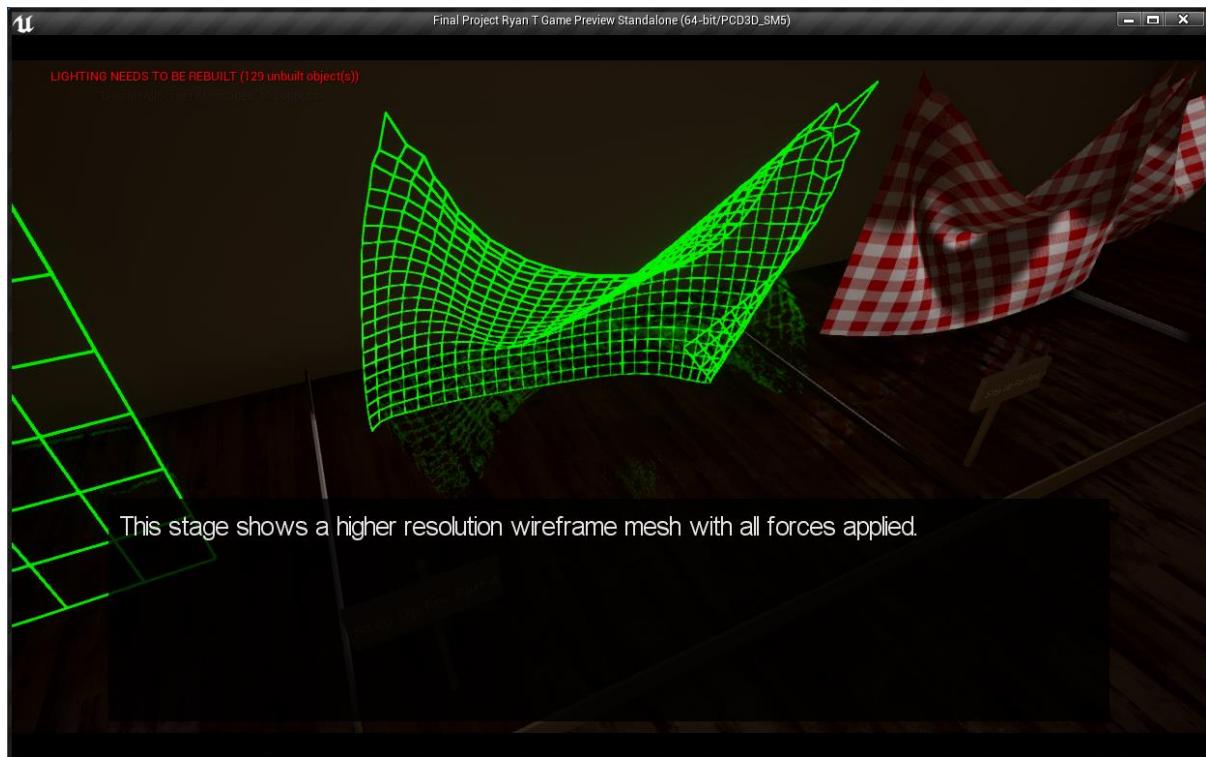


Figure 327

Added text to cloth demo object 5 to begin sequencing.

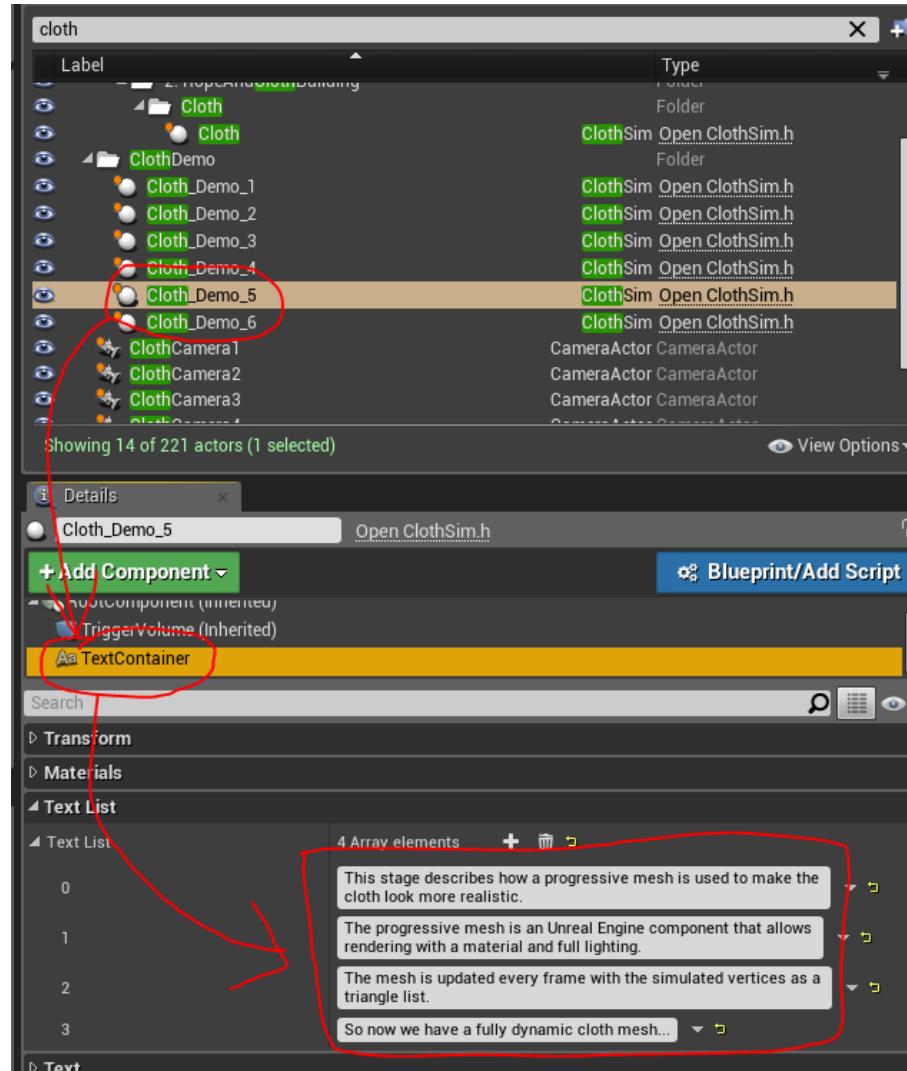


Figure 328

Setup sequence tracks cloth demo 5. Includes Render Mode & Text Container Index tracks. These are all exposed in code in the cloth code.

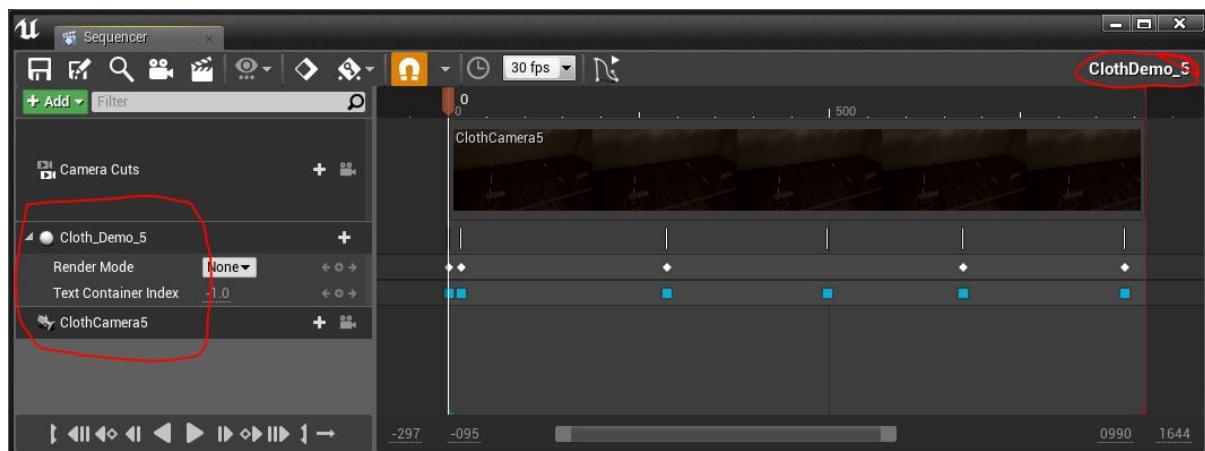
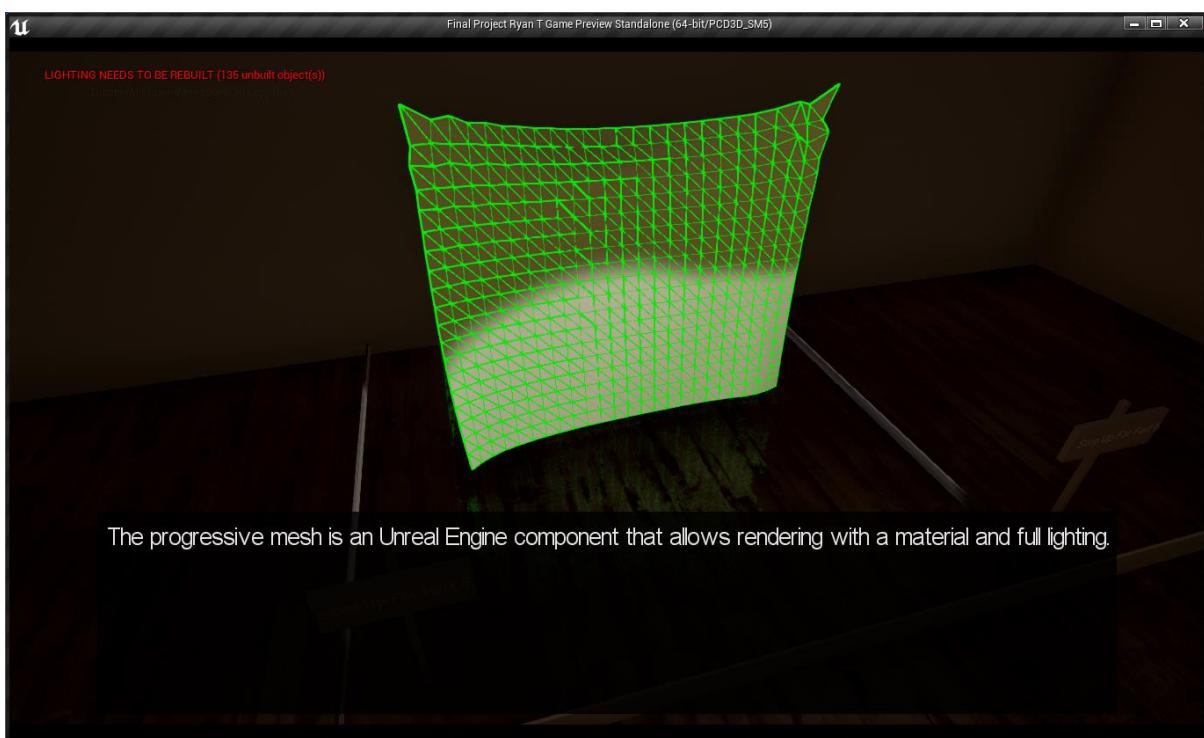
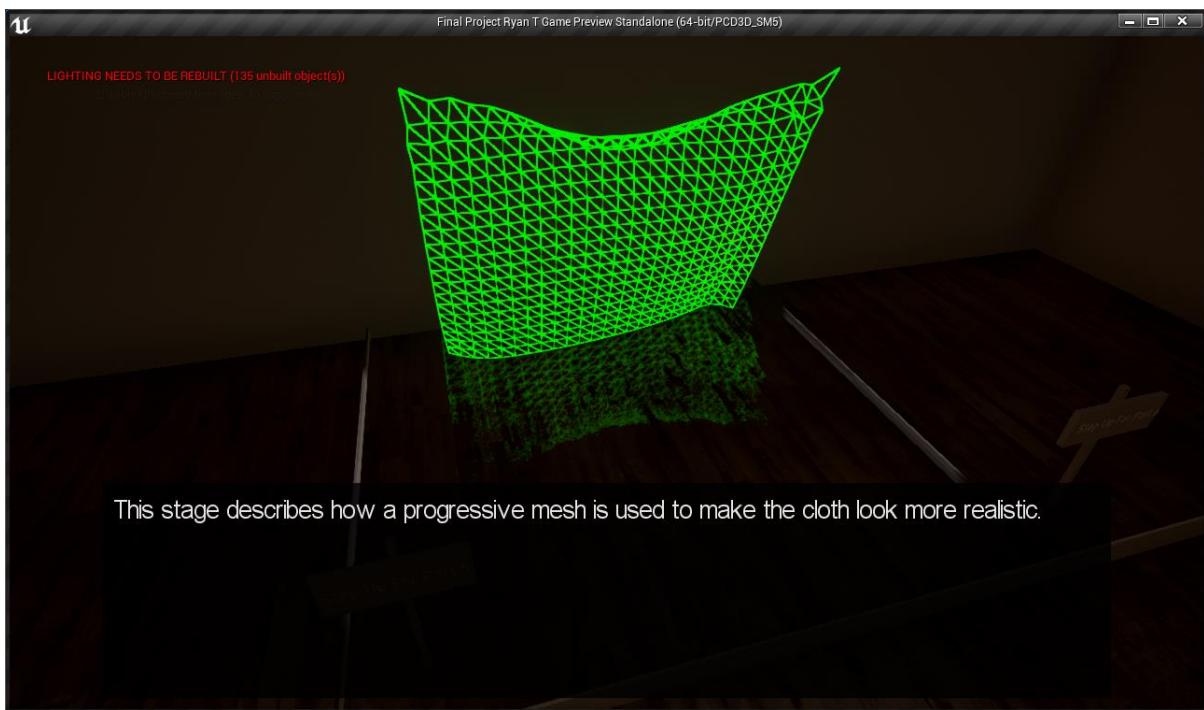


Figure 329

Finished sequence 5.



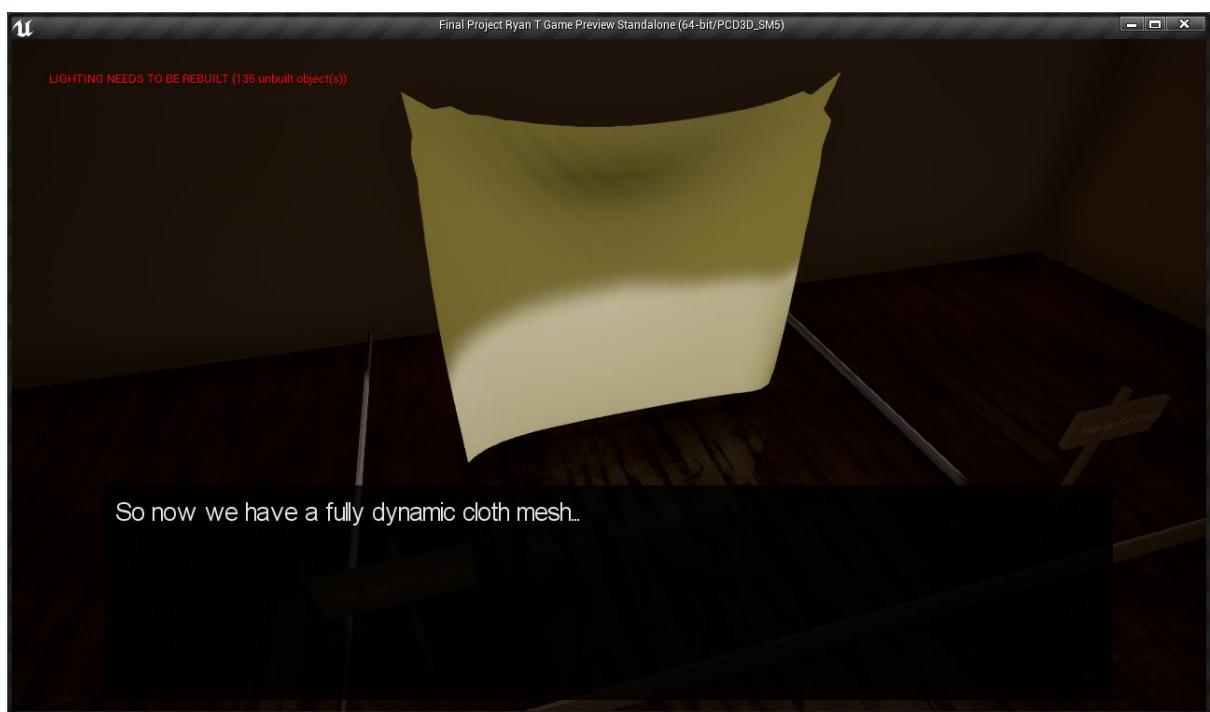
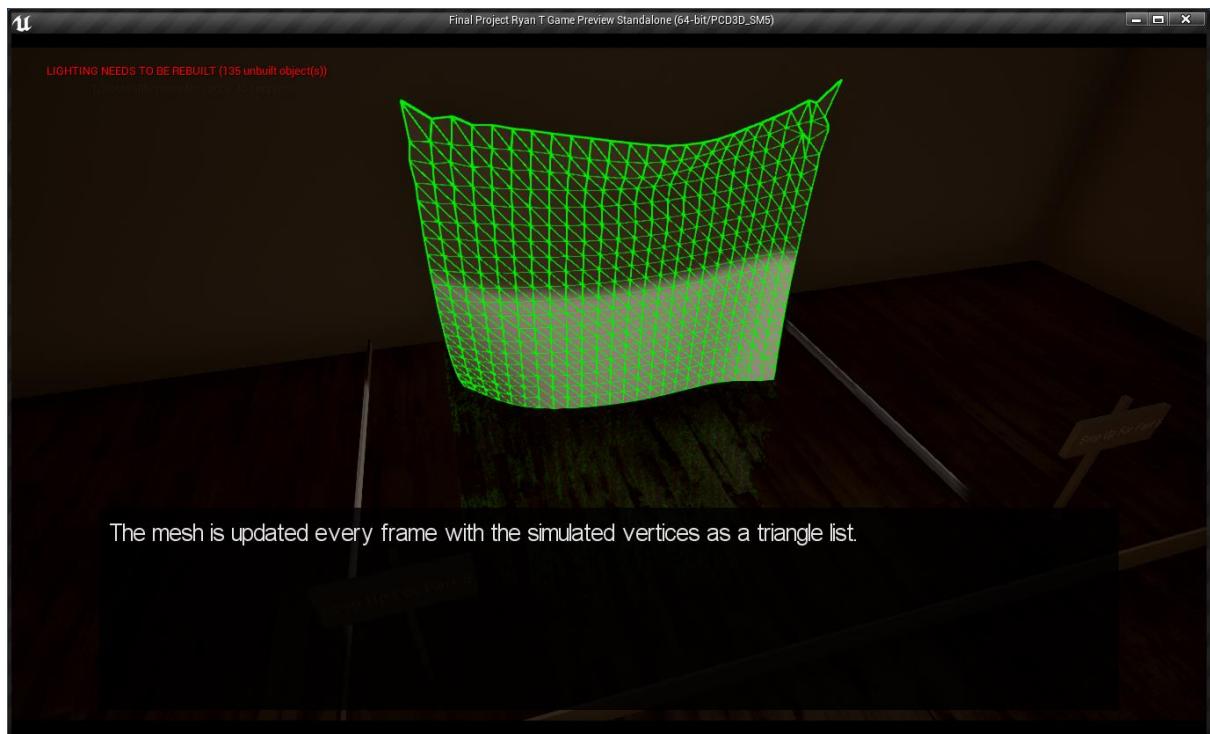


Figure 330 – Finished Sequence 5 (4 Images)

Added text to cloth demo object 6 to begin sequencing.

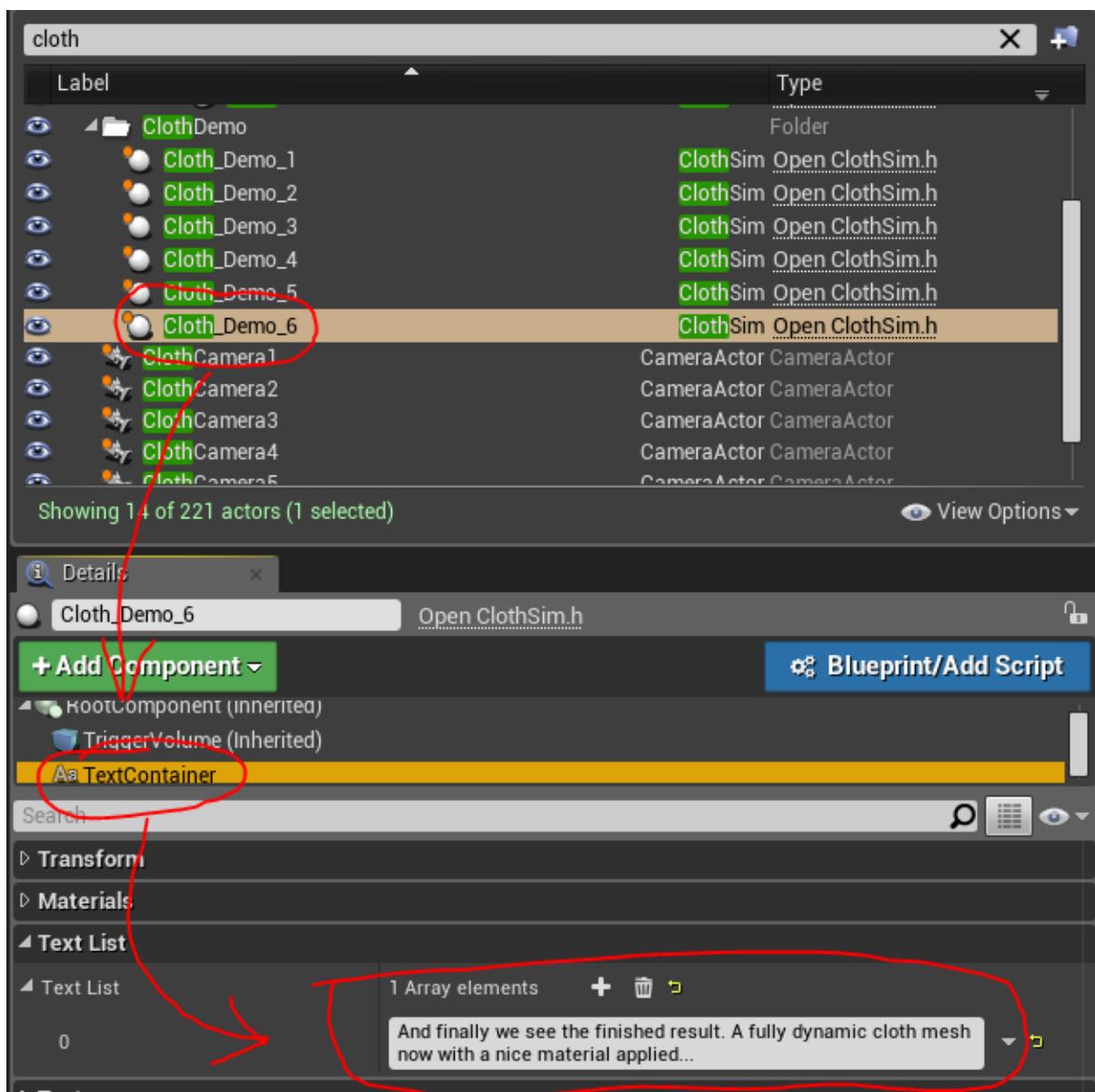


Figure 331

Setup sequence tracks cloth demo 6. Includes Render Mode & Text Container Index tracks. These are all exposed in code in the cloth code.

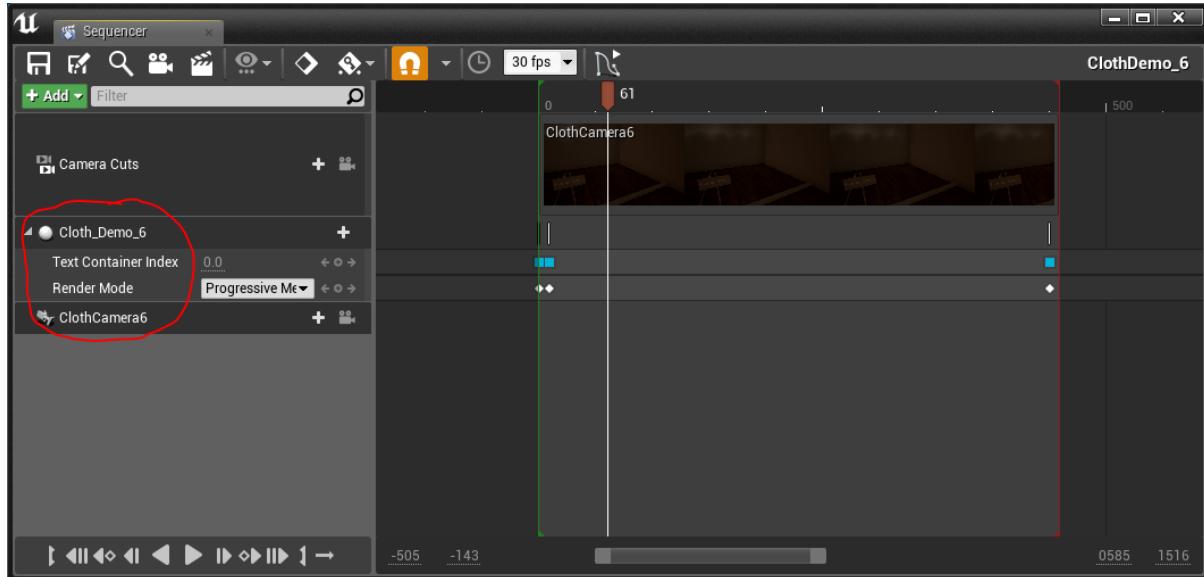


Figure 332

Finished sequence 6.

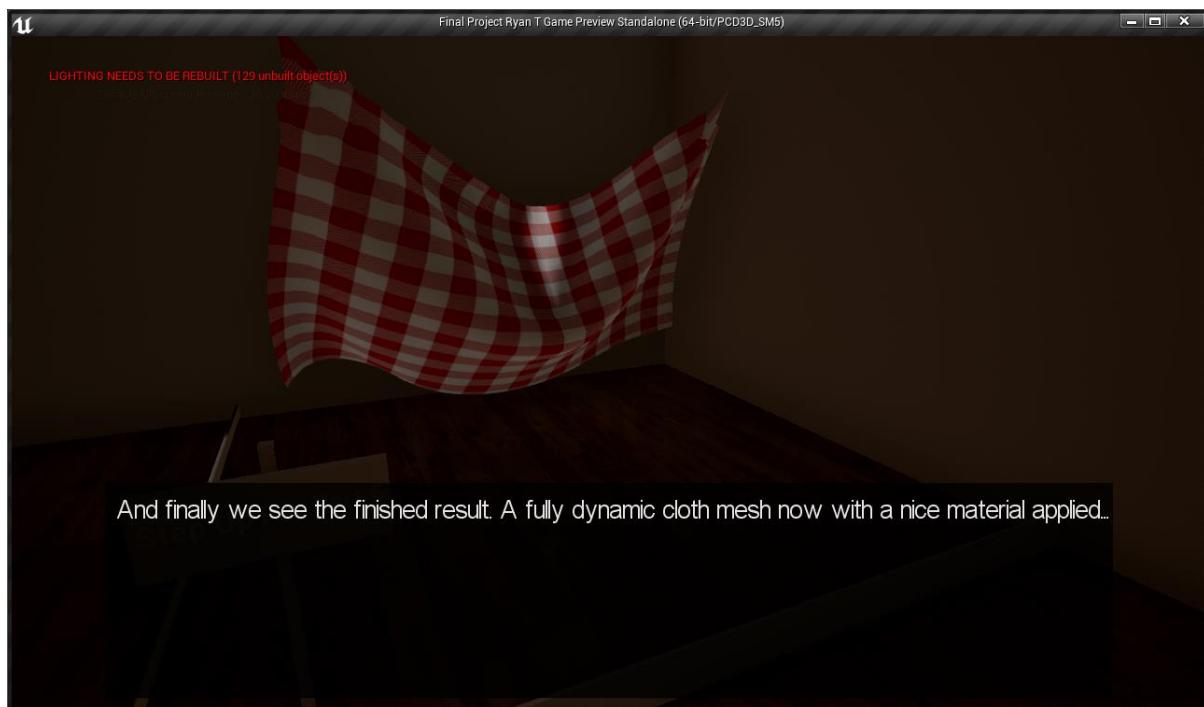


Figure 333 – Sequence 6

Added 2 new cloth render modes. Mesh and Wire square and Mesh and Wire triangle.

```
22 enum class eRenderMode : uint8
23 {
24     None,
25     WireframeSquares,
26     WireframeTriangles,
27     WireframeConstraints,
28     WireframeSquaresAndConstraints,
29     ProgressiveMesh,
30     ProgressiveMeshSquares,
31     ProgressiveMeshTriangles
32 };
33
```

Figure 334

Added code to cloth render function to support the 2 new modes.

```
364 // ****
365 // Render - Renders the cloth
366 // ****
367 void AClothSim::Render()
368 {
369     switch ( RenderMode )
370     {
371         case eRenderMode::None:
372             break;
373
374         case eRenderMode::WireframeTriangles:
375         {
376             DrawWireframeTriangles( bRenderModeIncremental );
377             break;
378         }
379
380         case eRenderMode::WireframeSquares:
381         {
382             DrawWireframeSquares( bRenderModeIncremental );
383             break;
384         }
385
386         case eRenderMode::WireframeConstraints:
387         {
388             DrawWireframeConstraints( bRenderModeIncremental );
389             break;
390         }
391
392         case eRenderMode::WireframeSquaresAndConstraints:
393         {
394             DrawWireframeSquares( false /*bRenderModeIncremental*/ );
395             DrawWireframeConstraints( bRenderModeIncremental );
396             break;
397         }
398
399         case eRenderMode::ProgressiveMesh:
400         {
401             UpdateMesh();
402             break;
403         }
404
405         case eRenderMode::ProgressiveMeshSquares:
406         {
407             UpdateMesh();
408             DrawWireframeSquares( bRenderModeIncremental );
409             break;
410         }
411
412         case eRenderMode::ProgressiveMeshTriangles:
413         {
414             UpdateMesh();
415             DrawWireframeTriangles( bRenderModeIncremental );
416             break;
417         }
418     }
419 }
```

Figure 335

Added 15 ropes into scene.

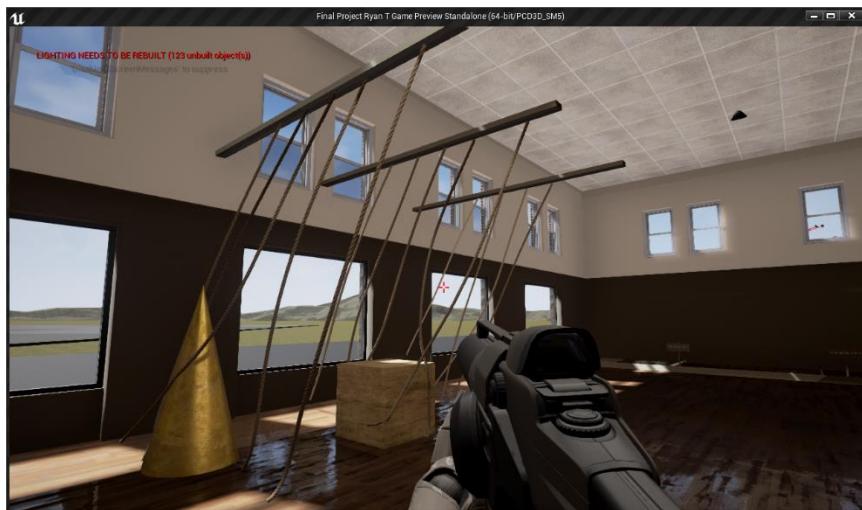


Figure 336

Added a TextContainer with text needed for rope demo to rope #1,

A screenshot of the Unreal Engine interface. The World Outliner shows a folder named 'Ropes' containing 15 entries labeled 'Rope1' through 'Rope15'. The 'Rope1' entry is selected and highlighted with a red circle. The Details panel shows the component tree for 'Rope1', with 'RootComponent (Inherited)' and 'TriggerVolume (Inherited)' expanded. A 'TextContainer' component is added, highlighted with a red circle. The 'TextList' section of the Details panel contains three array elements with the following text:

- These ropes use the same technique as the cloth - Verlet Integration with constraints.
- With one difference, the vertices are not in a grid layout, they are in a linear layout along the rope.
- The generated mesh is a tube along the rope with its centre being each of the simulated vertices.

Figure 337

Move rope #1 trigger volume to the correct place.

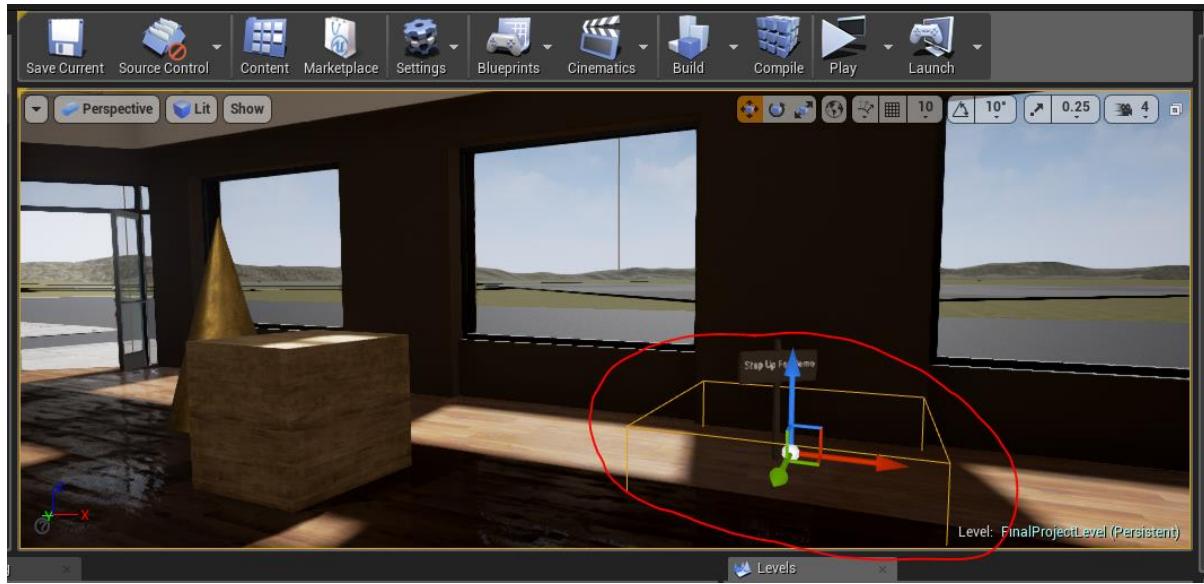


Figure 338

Created an empty level sequence asset and added it to rope #1 to be triggered when entering volume.

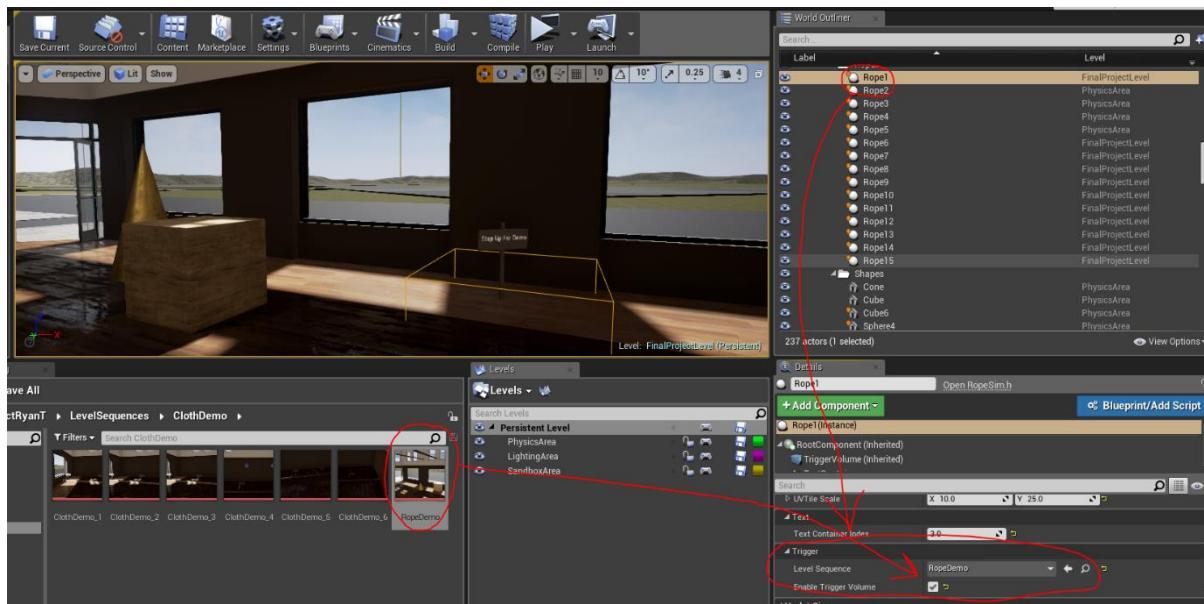


Figure 339

Added tracks to rope level sequence to move a large bubble from door, around ropes then out of window. Also displays text as with other sequences.

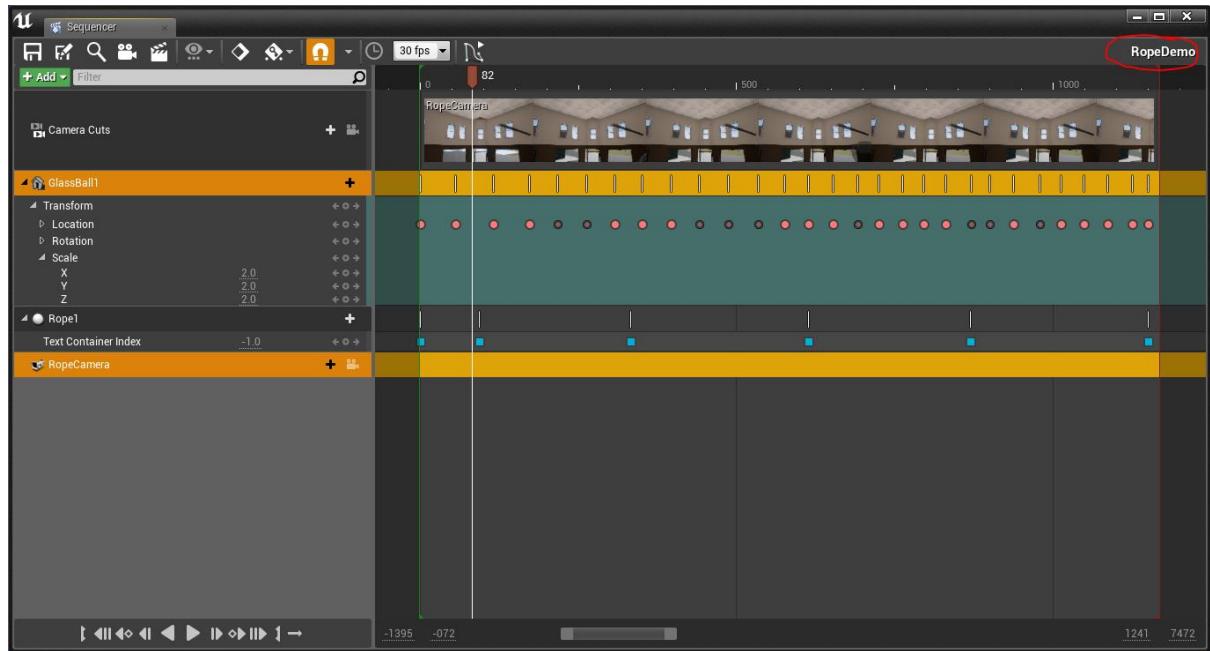


Figure 340

Shows editor viewport with bubble movement track highlighted and the camera that tracks it.

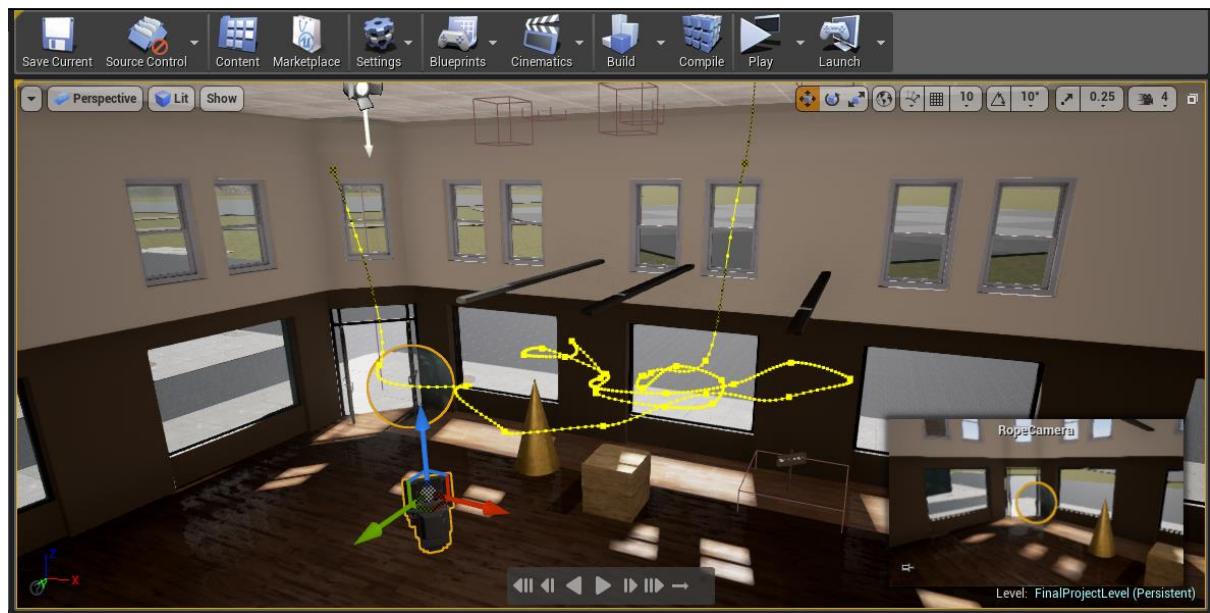
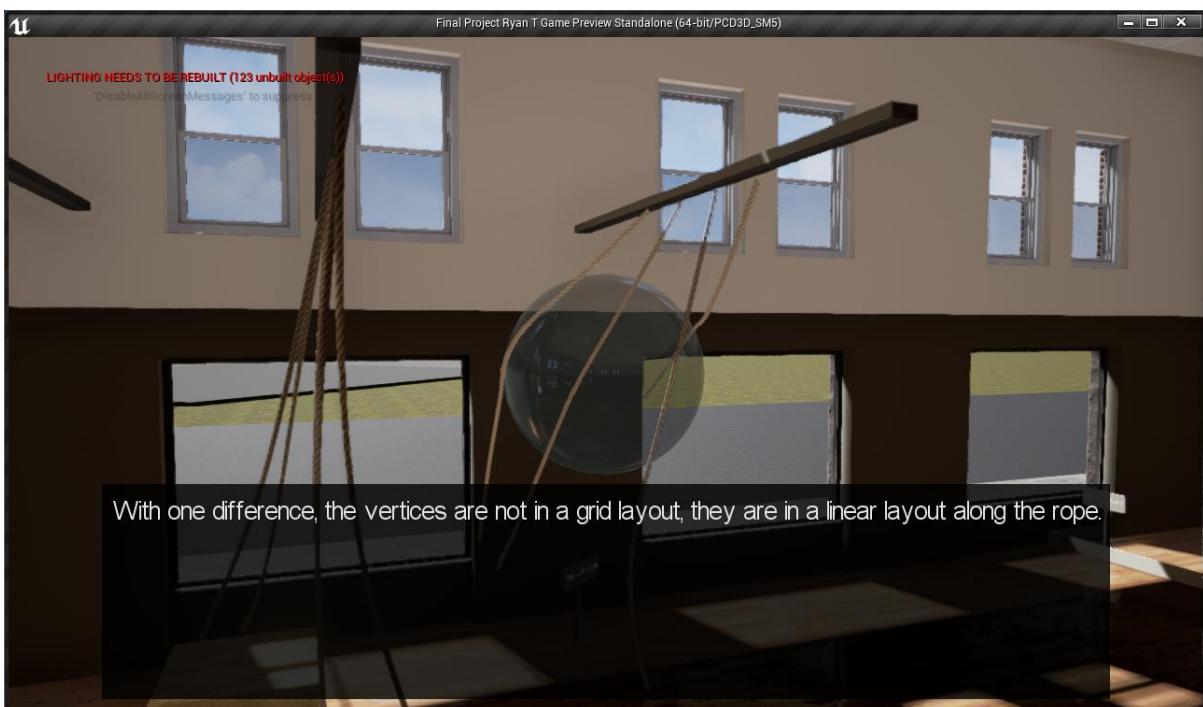


Figure 341

Rope Sequence finished.



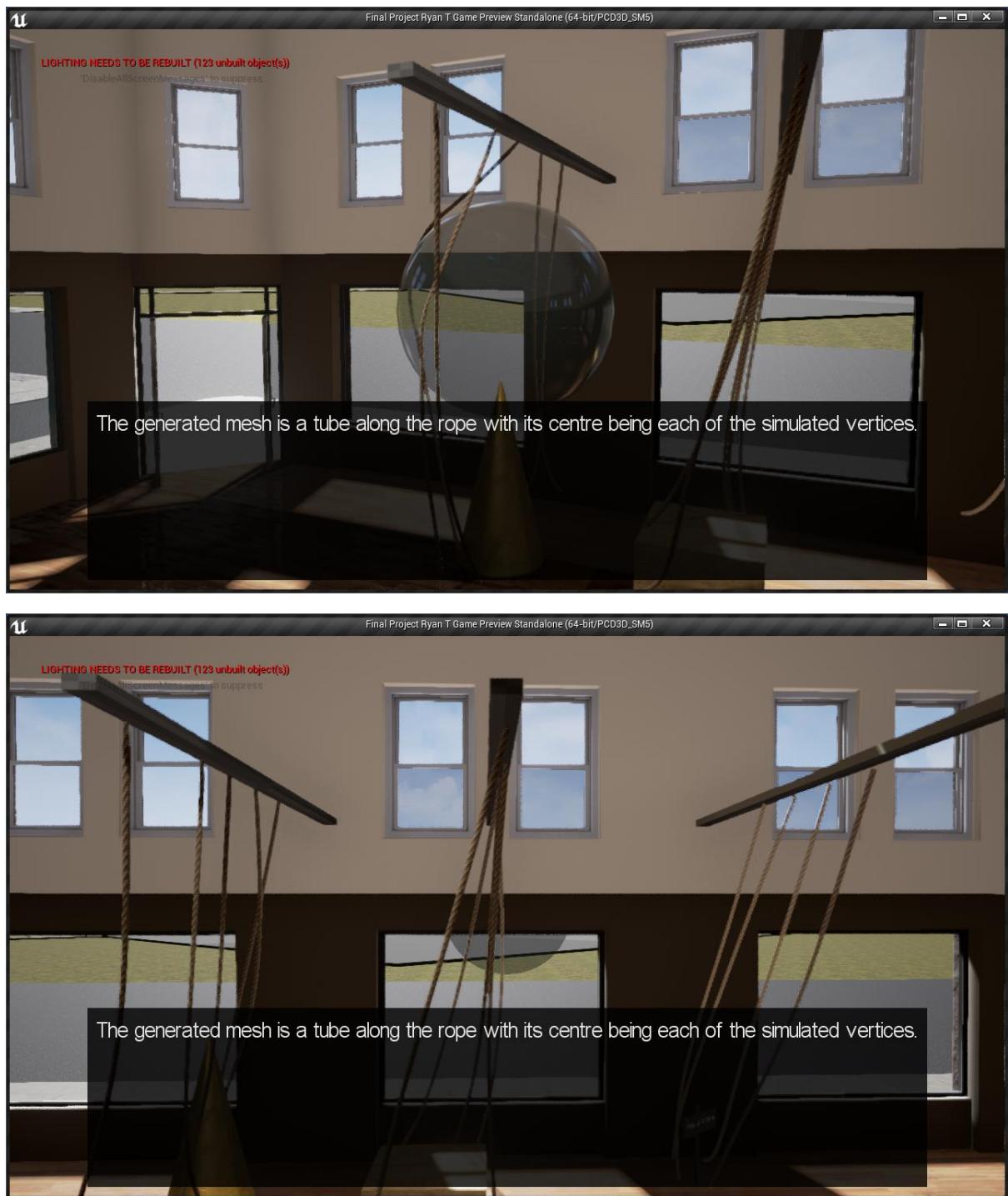


Figure 342 – Rope Demonstration (4 Images)

Added float UPROPERTY MatTickRange to VerletSim class so a maximum range for tick can be specified.

```

216     UPROPERTY(EditAnywhere, Category = "Verlet Settings|Trigger")
217     bool bEnableTriggerVolume = false;
218
219
220     UPROPERTY(EditAnywhere)
221     UBxComponent* TriggerVolume = nullptr;
222
223     UPROPERTY(EditAnywhere, Category = "Verlet Settings")
224     float MaxTickRange = 500.0f;
225
226 protected:
227
228     void SimulateVert(Verlet& Vert, float DeltaTime);
229     bool ResolveVertCollision(Verlet& Vert);
230     void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
231     void DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode, FColor Colour = FColor::Blue
232     void UpdateSubTitleText();
233

```

Figure 343

Added function OutsideOfTickRange that returns a bool. True when the VerletSim actor is more than MaxTickRange away from player else false.

```

234     void OnEndSequenceEvent();
235
236     UFUNCTION()
237     void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
238     UFUNCTION()
239     void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
240
241     bool OutsideOfTickRange()
242     {
243         if ( Timer < 10.0f )           // Must be allowed to Tick for at least 10 seconds
244             false;                  // to allow everything to settle
245
246         FVector Pos = GetWorld()->GetFirstPlayerController()->GetPawn()->GetActorLocation();
247         return FVector::Dist(Pos, GetActorLocation()) > MaxTickRange;
248     }
249
250
251

```

Figure 344

Added call to the function in Cloth.

```

59     // ****
60     // Tick - Called every frame
61     // ****
62     void AClothSim::Tick(float DeltaTime)
63     {
64         //DeltaTime /= 10.0f;          // Slow Motion
65
66         // GEngine->AddOnScreenDebugMessage(0, 0.0f, FColor::White, GetName() );
67
68         // DrawDebugString(GetWorld(), GetActorLocation(), GetName(), nullptr, FColor::White, 0.0f, true);
69
70         Super::Tick(DeltaTime);
71
72         if ( OutsideOfTickRange() )
73             return;
74
75         UpdateLockedVerts();
76         Simulate(DeltaTime);
77
78         for (int i = 0; i < NumConstrainIterations; i++)
79         {
80             Constrain();
81             ResolveCollision();
82         }
83
84         Render();
85     }
86

```

Figure 345

Added call to the function in Rope.

```
46 // ****
47 // Tick - Called every frame
48 // ****
49 void ARopeSim::Tick(float DeltaTime)
50 {
51     Super::Tick(DeltaTime);
52
53     if ( OutsideOfTickRange() )
54         return;
55
56     UpdateLockedVerts();
57     Simulate(DeltaTime);
58
59     for (int i = 0; i < NumConstrainIterations; i++)
60     {
61         Constrain();
62         ResolveCollision();
63     }
64
65     Render();
66 }
67
```

Figure 346

#### 6.4. Flocking Simulation Area (Physics Area)

Creating the new class that handles the whole simulation.

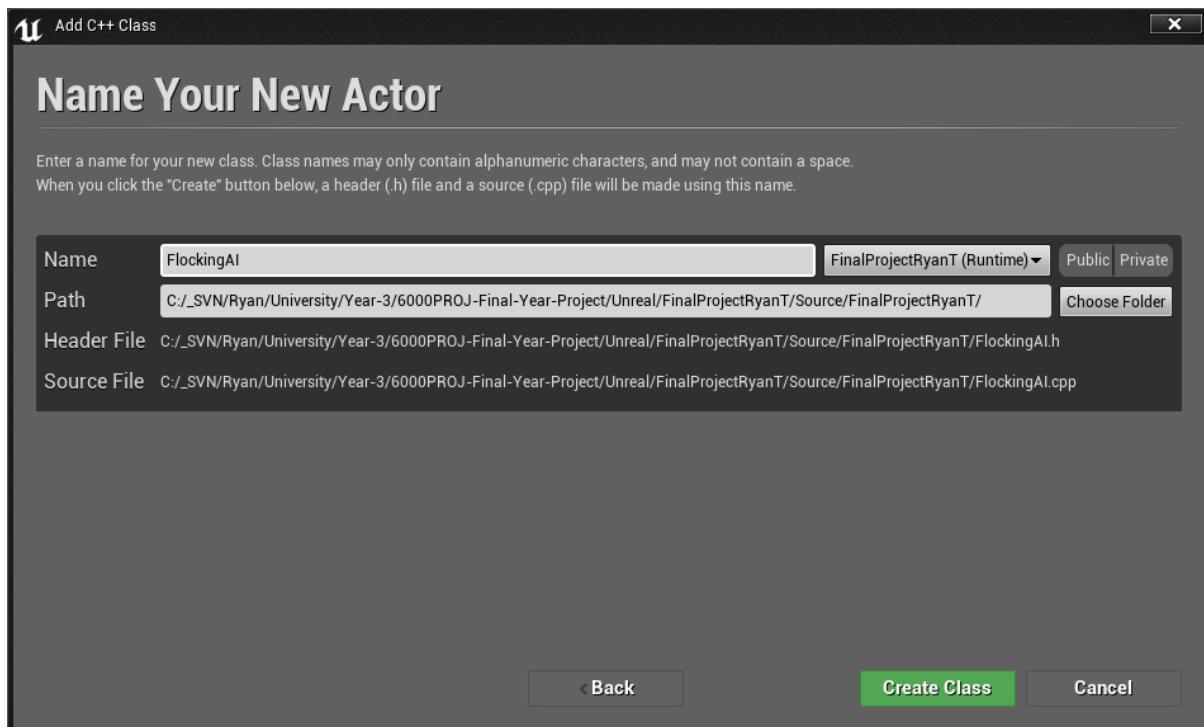


Figure 347

### Initial code (Header file)

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "FlockingAI.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API AFlockingAI : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     AFlockingAI();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23
24
25 };
26
```

Figure 348

### Initial code (Cpp file)

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "FlockingAI.h"
5
6
7 // ****
8 // Constructor
9 // ****
10 AFlockingAI::AFlockingAI()
11 {
12     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
13     PrimaryActorTick.bCanEverTick = true;
14 }
15
16 // ****
17 // BeginPlay - Called when the game starts or when spawned
18 // ****
19 void AFlockingAI::BeginPlay()
20 {
21     Super::BeginPlay();
22 }
23
24
25 // ****
26 // Tick - Called every frame
27 // ****
28 void AFlockingAI::Tick( float DeltaTime )
29 {
30     Super::Tick( DeltaTime );
31 }
32
33
34
35
```

Figure 349g

Adding values to header in prep for code.

```
UCLASS()
class FINALPROJECTRYANT_API AFlockingAI : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFlockingAI();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    FVector BirdPos;
    FVector BirdDirection;
    FVector BirdWaypoint;
    float BirdSpeed;
};
```

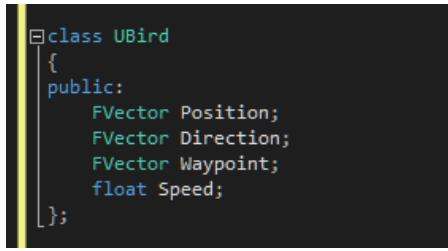
Figure 350

First stages of coding, adding temp location values, drawing bird and making it go to waypoint.

```
7
8 // ****
9 // Constructor
10 // ****
11 AFlockingAI::AFlockingAI()
12 {
13     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
14     PrimaryActorTick.bCanEverTick = true;
15
16     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
17     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
18
19 }
20
21 // ****
22 // BeginPlay - Called when the game starts or when spawned
23 // ****
24 void AFlockingAI::BeginPlay()
25 {
26     Super::BeginPlay();
27
28     BirdPos = FVector (-25880.000000, 39910.000000, 362.000122);           // First building
29     BirdDirection = FVector(-1, 0, 0);
30     BirdWaypoint = FVector(-38770.000000, 39260.000000, 644.334167);      // Fountain
31     BirdSpeed = 400.0f;
32 }
33
34
35 // ****
36 // Tick - Called every frame
37 // ****
38 void AFlockingAI::Tick( float DeltaTime )
39 {
40     Super::Tick( DeltaTime );
41
42     BirdPos += BirdDirection * BirdSpeed * DeltaTime;
43
44     FVector Dir = BirdWaypoint - BirdPos;
45     Dir.Normalize();
46
47     BirdDirection += Dir * 0.3f;
48
49     BirdDirection.Normalize();
50
51     DrawDebugDirectionalArrow(GetWorld(), BirdPos, BirdPos + BirdDirection * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f);
52     DrawDebugDirectionalArrow(GetWorld(), BirdPos, BirdPos + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f);
53
54     DrawDebugString(GetWorld(), BirdPos, GetName(), nullptr, FColor::White, 0.0f, true);
55
56 }
57
```

Figure 351

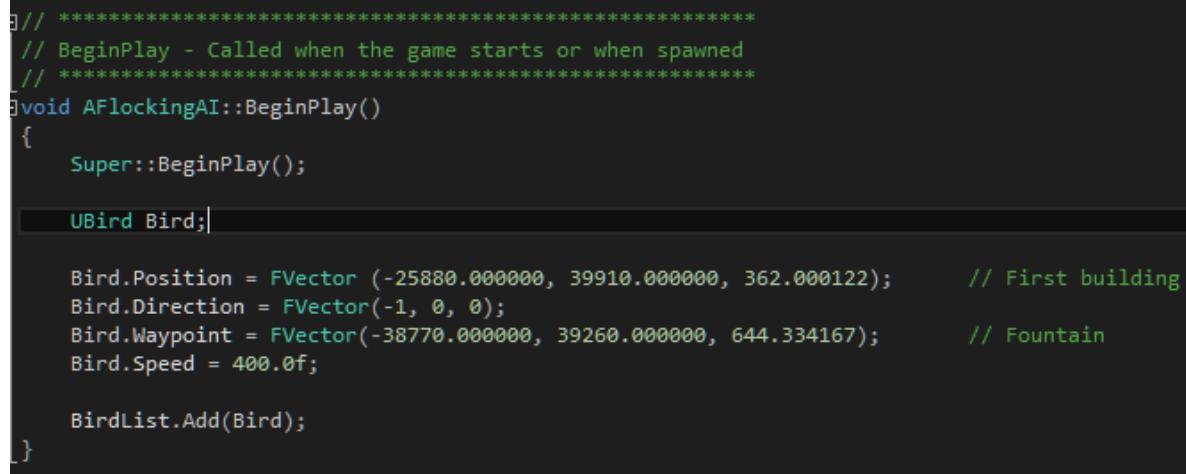
Moved values in header to new class in header.



```
class UBird
{
public:
    FVector Position;
    FVector Direction;
    FVector Waypoint;
    float Speed;
};
```

Figure 352

Created new instance of Bird class and added it to BirdList.



```
// ****
// BeginPlay - Called when the game starts or when spawned
// ****
void AFlockingAI::BeginPlay()
{
    Super::BeginPlay();

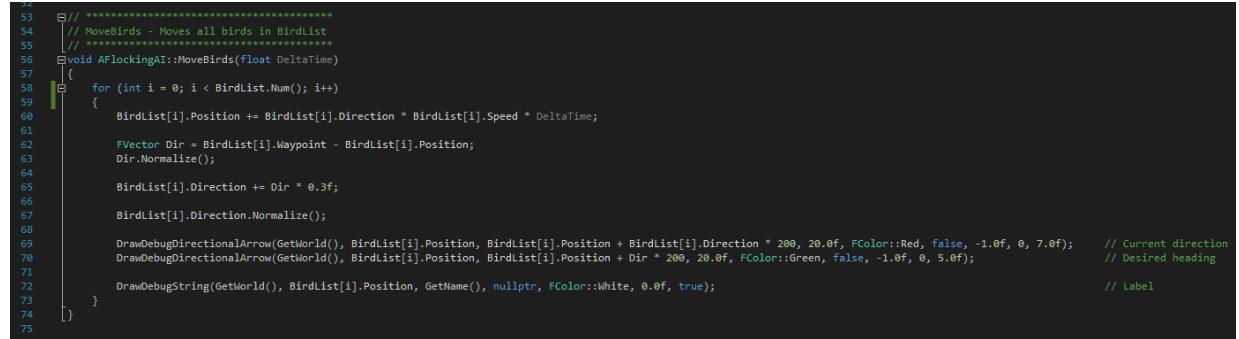
    UBird Bird;

    Bird.Position = FVector (-25880.000000, 39910.000000, 362.000122);           // First building
    Bird.Direction = FVector(-1, 0, 0);
    Bird.Waypoint = FVector(-38770.000000, 39260.000000, 644.334167);          // Fountain
    Bird.Speed = 400.0f;

    BirdList.Add(Bird);
}
```

Figure 353

Moved temp code from tick to its own function.



```
53 // ****
54 // MoveBirds - Moves all birds in Birdlist
55 // ****
56 void AFlockingAI::MoveBirds(float DeltaTime)
57 {
58     for (int i = 0; i < BirdList.Num(); i++)
59     {
60         BirdList[i].Position += BirdList[i].Direction * BirdList[i].Speed * DeltaTime;
61
62         FVector Dir = BirdList[i].Waypoint - BirdList[i].Position;
63         Dir.Normalize();
64
65         BirdList[i].Direction += Dir * 0.3f;
66
67         BirdList[i].Direction.Normalize();
68
69         DrawDebugDirectionalArrow(GetWorld(), BirdList[i].Position, BirdList[i].Position + BirdList[i].Direction * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f); // Current direction
70         DrawDebugDirectionalArrow(GetWorld(), BirdList[i].Position, BirdList[i].Position + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f); // Desired heading
71
72         DrawDebugString(GetWorld(), BirdList[i].Position, GetName(), nullptr, FColor::White, 0.0f, true); // Label
73     }
74 }
```

Figure 354

Gameplay - bird flies to waypoint, green arrow shows direction to waypoint and red arrow is current direction.

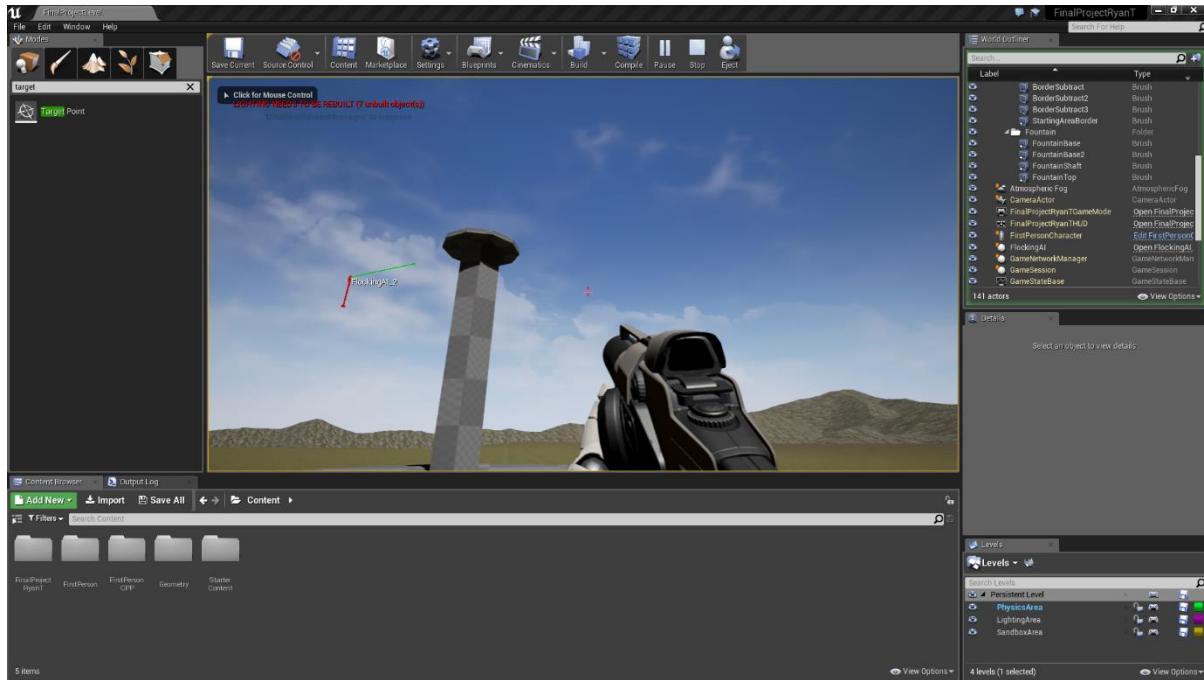


Figure 355

Added target points to act as waypoints.

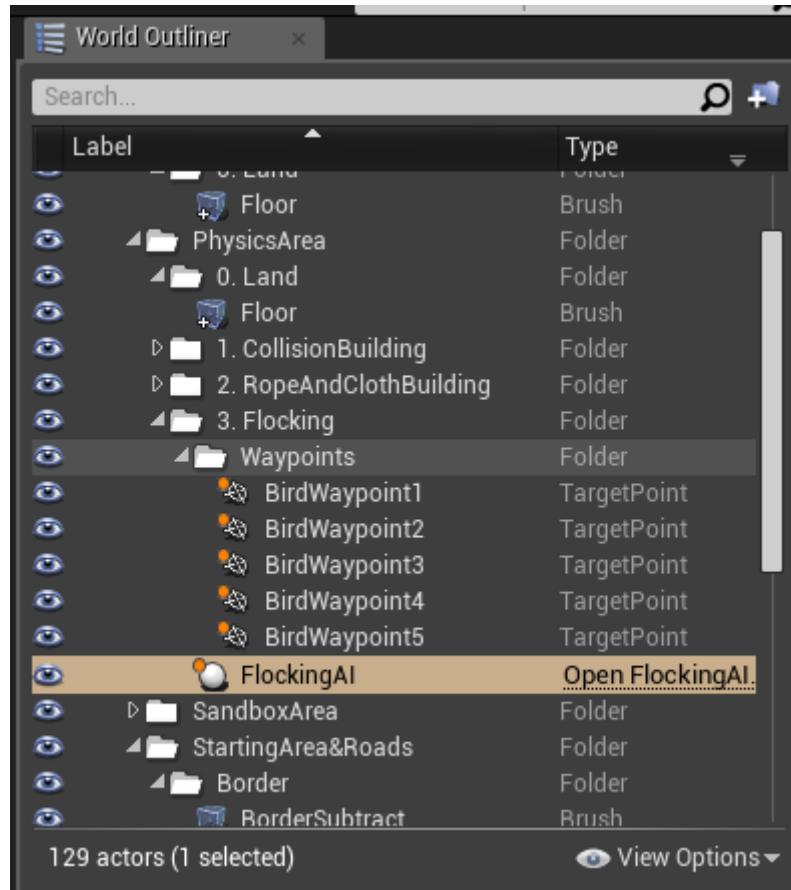


Figure 356

Made an array of the waypoints in the Flocking header file.

```
32
33     TArray<UBird> BirdList;
34
35     UPROPERTY(EditAnywhere)
36     TArray<ATargetPoint*> Waypoints;
37
```

Figure 357

Array in the flockingAI actor in the scene, filled the array with the waypoints placed in the level.

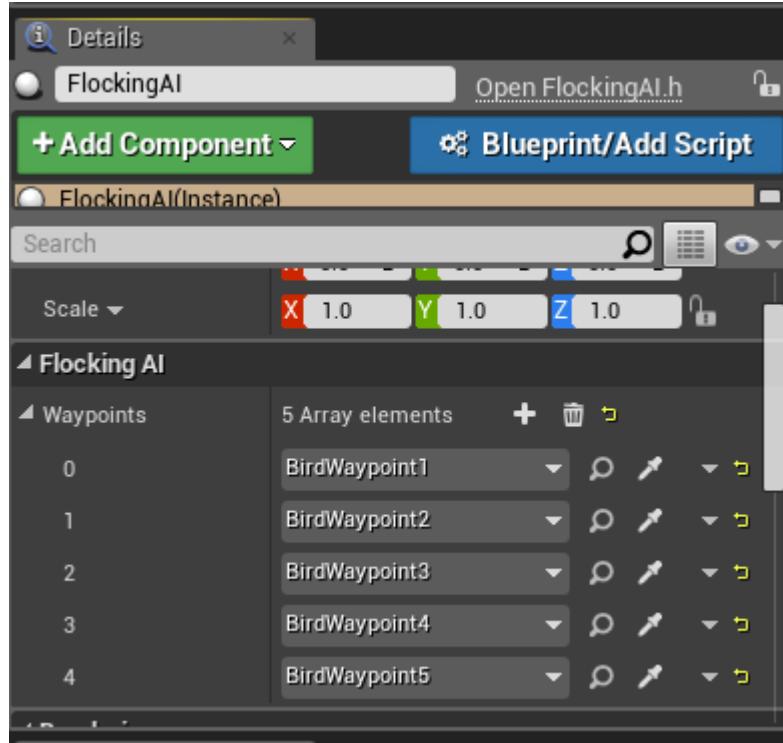


Figure 358

Added temp code to add birds near waypoints at random points.

```
19
20     // ****
21     // BeginPlay - Called when the game starts or when spawned
22     // ****
23     void AFlockingAI::BeginPlay()
24     {
25         Super::BeginPlay();
26
27         UBird Bird;
28
29         for (int i = 0; i < Waypoints.Num(); i++)
30         {
31             FVector Offset(FMath::FRandRange(-2000, 2000), FMath::FRandRange(-3000, 3000), FMath::FRandRange(-200, 200));
32
33             Bird.Position = Waypoints[i]->GetActorLocation() + Offset;           // Waypoint location plus rand offset
34             Bird.Direction = FVector(-1, 0, 0);
35             Bird.Waypoint = Waypoints[i]->GetActorLocation();                  // Waypoint location
36             Bird.Speed = 400.0f;
37
38             BirdList.Add(Bird);
39         }
40     }
```

Figure 359

Added code to draw a sphere around each waypoint, this is called in tick.

```
77     // ****
78     // ShowWaypoints - Draws a debug shpere around waypoint location
79     // ****
80     void AFlockingAI::ShowWaypoints()
81     {
82         for (int i = 0; i < Waypoints.Num(); i++)
83         {
84             DrawDebugSphere(GetWorld(), Waypoints[i]->GetActorLocation(), 100.0f, 8, FColor::Red, false, -1.0f, 0, 5.0f);
85         }
86     }
87 
```

Figure 360

New function to select a random waypoint.

```
91     // ****
92     // PickWaypoint - Picks a new waypoint, ignoring the previous one
93     // ****
94     FVector AFlockingAI::PickNewWaypoint(FVector OldWaypoint)
95     {
96         int Index = FMath::RandRange(0, Waypoints.Num() - 1); // Index is a random number between 0 and the number of waypoints minus 1.
97
98         // If the distance between the old waypoint and the new waypoint
99         // is less than 10, then skip it and get next waypoint.
100        if ( FVector::Dist(Waypoints[Index]->GetActorLocation(), OldWaypoint) < 10.0f )
101            Index = (Index + 1) % Waypoints.Num(); // Get next index, making sure it stays within the bounds of the array
102
103        return Waypoints[Index]->GetActorLocation(); // (%Modulo operator) divides left by right, then subtracts the remainder from this operation from the right.
104    }
105 } 
```

Figure 361

Called the new function in the MoveBirds function. Birds now fly around picking random waypoints.

```
54     // ****
55     // MoveBirds - Moves all birds in BirdList
56     // ****
57     void AFlockingAI::MoveBirds(float DeltaTime)
58     {
59         for (int i = 0; i < BirdList.Num(); i++)
60         {
61             BirdList[i].Position += BirdList[i].Direction * BirdList[i].Speed * DeltaTime;
62
63             FVector Dir = BirdList[i].Waypoint - BirdList[i].Position;
64             Dir.Normalize();
65
66             BirdList[i].Direction += Dir * 0.15f;
67
68             BirdList[i].Direction.Normalize();
69
70             if (FVector::Dist(BirdList[i].Position, BirdList[i].Waypoint) < 200.0f) // IF bird is within 2 metres of waypoint...
71             {
72                 BirdList[i].Waypoint = PickNewWaypoint(BirdList[i].Waypoint); // Pick new waypoint,
73             }
74
75             DrawDebugDirectionalArrow(GetWorld(), BirdList[i].Position, BirdList[i].Position + BirdList[i].Direction * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f); // Current direction
76             DrawDebugDirectionalArrow(GetWorld(), BirdList[i].Position, BirdList[i].Position + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f); // Desired heading
77
78             FString S;
79             S.Append("Bird : "); // Name each bird
80             S.AppendInt(i + 1);
81
82             DrawDebugString(GetWorld(), BirdList[i].Position, S, nullptr, FColor::White, 0.0f, true); // Label
83         }
84     } 
```

Figure 362

New function to find neighbouring birds.

```
127
128     // ****
129     // GetNeighbours - Returns a list of pointers to neighbouring birds
130     // ****
131     TArray<UBird*> AFlockingAI::GetNeighbours(UBird& Bird)
132     {
133         TArray<UBird*> Neighbours;
134
135         for (int i = 0; i < BirdList.Num(); i++)
136         {
137             if (&BirdList[i] == &Bird)
138                 continue;
139
140             if (FVector::Dist(BirdList[i].Position, Bird.Position) < GroupingDistance)
141             {
142                 Neighbours.Add(&BirdList[i]);
143             }
144         }
145
146         return Neighbours;
147     }
148
```

Figure 363

New function to get the direction to the centre of a flocking group of birds.

```
148
149     // ****
150     // GetDirectionToGroup - Returns normalised direction to the centre of the group
151     // ****
152     FVector AFlockingAI::GetDirectionToGroup(UBird& CurrentBird, TArray<UBird*>& Neighbours)
153     {
154         FVector DirectionToGroup(0.0f, 0.0f, 0.0f);
155
156         if (Neighbours.Num())
157         {
158             FVector GroupCentre(0.0f, 0.0f, 0.0f);
159
160             for (int j = 0; j < Neighbours.Num(); j++)
161             {
162                 GroupCentre += Neighbours[j]->Position;
163             }
164
165             GroupCentre /= float(Neighbours.Num());
166
167             DirectionToGroup = GroupCentre - CurrentBird.Position;
168             DirectionToGroup.Normalize();
169         }
170
171         return DirectionToGroup;
172     }
```

Figure 364

These two new functions are called in MoveBirds, birds now group together in flocks.

```

62 // ****
63 // MoveBirds - Moves all birds in Birdlist
64 // ****
65 void AFlockingAI::MoveBirds(float DeltaTime)
66 {
67     for (int i = 0; i < BirdList.Num(); i++)
68     {
69         UBird& CurrentBird = BirdList[i];
70
71         TArray<UBird*> Neighbours = GetNeighbours(CurrentBird);
72
73         FVector DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);
74
75         FVector Dir = CurrentBird.Waypoint - CurrentBird.Position;
76         Dir.Normalize();
77         Dir += DirectionToGroup;
78
79         CurrentBird.Direction += Dir * 0.15f;
80         CurrentBird.Direction.Normalize();
81
82         CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;
83
84         if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 200.0f)    // If bird is within 2 metres of waypoint...
85         {
86             CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint);           // Pick new waypoint.
87         }
88
89
90         DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + CurrentBird.Direction * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f); // Current direction
91         DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f); // Desired heading
92
93         //FString S;
94         //S.Append("bird : ");
95         //S.AppendInt(i + 1);
96
97         DrawDebugString(GetWorld(), CurrentBird.Position, "", nullptr, FColor::White, 0.0f, true); // Label
98     }
99 }

```

Figure 365

Birds now flocking together when they get close to each other.

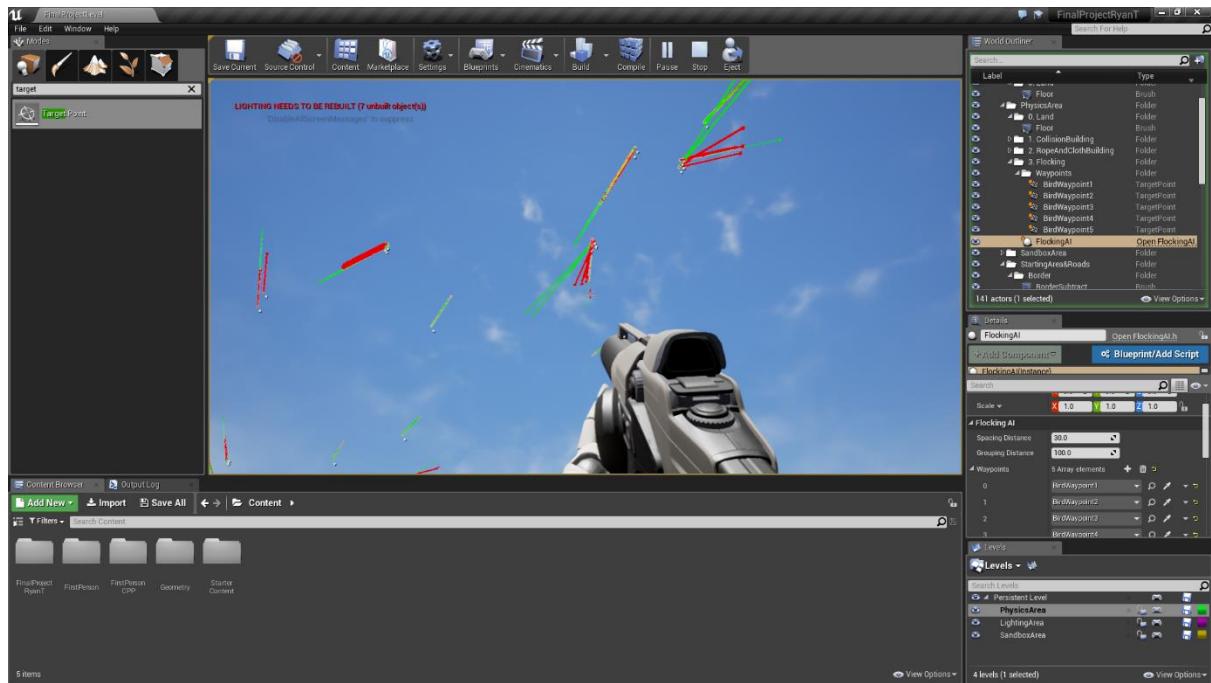


Figure 366

New function to get the direction away from neighbouring birds that are too close, to keep birds spaced apart.

```
200 // ****
201 // GetDirectionAwayFromNeighbour - Returns the force vector to move 'CurrentBird' away from its neighbours
202 // ****
203
204 FVector AFlockingAI::GetDirectionAwayFromNeighbour(UBird& CurrentBird, TArray<UBird*>& Neighbours)
205 {
206     FVector DirectionAwayFromNeighbour(0.0f, 0.0f, 0.0f);
207
208     for (int j = 0; j < Neighbours.Num(); j++)
209     {
210         FVector Direction = CurrentBird.Position - Neighbours[j]->Position;
211
212         float Dist = Direction.Size();
213
214         if (Dist < SpacingDistance)
215         {
216             // TODO: Potential divide by zero (If birds are in exactly the same place)
217
218             Direction /= Dist;           // Normalise
219
220             // TODO: Add more force the nearer we are to the neighbouring bird (weighting)
221
222             DirectionAwayFromNeighbour += Direction;
223         }
224     }
225
226     return DirectionAwayFromNeighbour;
227 }
```

Figure 367

New function to get average direction of neighbouring birds to make them face a common direction.

```
228 // ****
229 // GetAlignmentDirection - Returns the force vector align 'CurrentBird' with it's neighbours
230 // ****
231
232 FVector AFlockingAI::GetAlignmentDirection(UBird& CurrentBird, TArray<UBird*>& Neighbours)
233 {
234     FVector Alignment(0.0f, 0.0f, 0.0f);
235
236     if ( Neighbours.Num() )
237     {
238         for (int j = 0; j < Neighbours.Num(); j++)
239             Alignment += Neighbours[j]->Direction;
240         Alignment /= float(Neighbours.Num());
241     }
242
243 }
```

Figure 368

Two new functions called in MoveBirds, birds in groups are now nicely spaced, and face the same general direction.

```

67 // ****
68 // MoveBirds - Moves all birds in BirdList
69 // ****
70 void AFlockingAI::MoveBirds(float DeltaTime)
71 {
72     for (int i = 0; i < BirdList.Num(); i++)
73     {
74         UBird& CurrentBird = BirdList[i];
75
76         TArray<UBird*> Neighbours = GetNeighbours(CurrentBird);
77
78         FVector DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);
79         FVector DirectionAwayFromNeighbour = GetDirectionAwayFromNeighbour(CurrentBird, Neighbours);
80         FVector AlignmentDirection = GetAlignmentDirection(CurrentBird, Neighbours);
81
82         FVector ToWaypoint = CurrentBird.Waypoint - CurrentBird.Position;
83         ToWaypoint.Normalize();
84
85         FVector Dir = ToWaypoint;
86
87         Dir += AlignmentDirection;
88         Dir += DirectionToGroup;
89         Dir += DirectionAwayFromNeighbour;
90
91         CurrentBird.Direction += Dir * 0.15f;
92
93         CurrentBird.Direction.Normalize();
94
95         CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;
96
97         if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 200.0f) // If bird is within 2 metres of waypoint...
98         {
99             CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint); // Pick new waypoint.
100        }
101
102    }
103
104    // DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + CurrentBird.Direction * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f); // Current direction
105    // DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f); // Desired heading
106
107    // FString S;
108    // S.Append("Bird : "); // Name each bird
109    // S.AppendInt(i + 1);
110
111    DrawDebugString(GetWorld(), CurrentBird.Position, "", nullptr, FColor::White, 0.0f, true); // Label
112 }

```

Figure 369

New function called in MoveBirds, birds now avoid projectiles.

```

67 // ****
68 // MoveBirds - Moves all birds in BirdList
69 // ****
70 void AFlockingAI::MoveBirds(float DeltaTime)
71 {
72     for (int i = 0; i < BirdList.Num(); i++)
73     {
74         UBird& CurrentBird = BirdList[i];
75
76         TArray<UBird*> Neighbours = GetNeighbours(CurrentBird);
77
78         FVector DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);
79         FVector DirectionAwayFromNeighbour = GetDirectionAwayFromNeighbour(CurrentBird, Neighbours);
80         FVector AlignmentDirection = GetAlignmentDirection(CurrentBird, Neighbours);
81         FVector ProjectileAvoid = GetProjectileAvoidDirection(CurrentBird);
82
83         FVector ToWaypoint = CurrentBird.Waypoint - CurrentBird.Position;
84         ToWaypoint.Normalize();
85
86         FVector Dir = ToWaypoint;
87
88         Dir += AlignmentDirection;
89         Dir += DirectionToGroup;
90         Dir += DirectionAwayFromNeighbour;
91
92         Dir += ProjectileAvoid;
93
94         CurrentBird.Direction += Dir * 0.15f;
95
96         CurrentBird.Direction.Normalize();
97
98         CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;
99
100        if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 200.0f) // If bird is within 2 metres of waypoint...
101        {
102            CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint); // Pick new waypoint.
103        }
104
105    }
106
107    // DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + CurrentBird.Direction * 200, 20.0f, FColor::Red, false, -1.0f, 0, 7.0f); // Current direction
108    // DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + Dir * 200, 20.0f, FColor::Green, false, -1.0f, 0, 5.0f); // Desired heading
109
110    // FString S;
111    // S.Append("Bird : "); // Name each bird
112    // S.AppendInt(i + 1);
113
114    DrawDebugString(GetWorld(), CurrentBird.Position, "", nullptr, FColor::White, 0.0f, true); // Label
115 }

```

Figure 370

New function to make birds avoid projectiles, uses a projectile list that is updated within FinalProjectRyanTProjectile.cpp

```
232     // ****
233     // GetProjectileAvoidDirection - Returns direction away from any close projectiles
234     // ****
235     FVector AFlockingAI::GetProjectileAvoidDirection(UBird& CurrentBird)
236     {
237         FVector ProjectileAvoid(0.0f, 0.0f, 0.0f);
238
239         for (int p = 0; p < AFinalProjectRyanTProjectile::ProjectileList.Num(); p++)
240         {
241             FVector ProjectilePos = AFinalProjectRyanTProjectile::ProjectileList[p]->GetActorLocation();
242             if (FVector::Dist(CurrentBird.Position, ProjectilePos) < AvoidProjectileDistance)
243             {
244                 FVector NewDir = CurrentBird.Position - ProjectilePos;
245                 NewDir.Normalize();
246                 ProjectileAvoid += NewDir;
247             }
248         }
249
250         return ProjectileAvoid;
251     }
```

Figure 371

Added uproperty to assign a skeletal mesh resource for the birds.

```
32
33     UCLASS()
34     class FINALPROJECTRYANT_API AFlockingAI : public AActor
35     {
36         GENERATED_BODY()
37
38     public:
39         // Sets default values for this actor's properties
40         AFlockingAI();
41
42         // Called when the game starts or when spawned
43         virtual void BeginPlay() override;
44
45         // Called every frame
46         virtual void Tick( float DeltaSeconds ) override;
47
48         TArray<UBird> BirdList;
49
50         UPROPERTY(EditAnywhere)
51         float SpacingDistance = 100.0f;
52
53         UPROPERTY(EditAnywhere)
54         float GroupingDistance = 200.0f;
55
56         UPROPERTY(EditAnywhere)
57         float AvoidProjectileDistance = 1200.0f;
58
59         UPROPERTY(EditAnywhere)
60         TArray<ATargetPoint*> Waypoints;
61
62         UPROPERTY(EditAnywhere)
63         USkeletalMesh* BirdMeshResource = nullptr;
64
65     private:
66         UPrimitiveComponent* LastAddedComponent = nullptr;
67
68     private:
```

Figure 372

Added code to create a skeletal mesh instance for each bird from said resource.

```
29
30     // ****
31     // BeginPlay - Called when the game starts or when spawned
32     // ****
33     void AFlockingAI::BeginPlay()
34     {
35         Super::BeginPlay();
36
37         int MaxBirds = 100;
38
39         UBird Bird;
40
41         for (int i = 0; i < MaxBirds; i++)
42         {
43             FVector Offset(FMath::FRandRange(-2000, 2000), FMath::FRandRange(-3000, 3000), FMath::FRandRange(-200, 200));
44
45             Bird.Position = GetActorLocation() + Offset;           // Waypoint location plus rand offset
46             Bird.Direction = FVector(-1, 0, 0);
47             Bird.Waypoint = PickNewWaypoint(Bird.Position);        // Pick new waypoint.
48             Bird.Speed = 400.0f;
49
50             // Create a skeletal mesh instance for this bird, using BirdMeshResource as a resource
51             Bird.BirdMesh = NewObject<USkeletalMeshComponent>(this, NAME_None);
52
53             Bird.BirdMesh->SetSkeletalMesh(BirdMeshResource);
54
55             Bird.UpdateTransform();
56
57             Bird.BirdMesh->RegisterComponent();
58
59             // KeepWorldTransform means that each component is not relative to the flockingAI actor, it has its own worldspace position
60             Bird.BirdMesh->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);
61
62             LastAddedComponent = Bird.BirdMesh;
63
64             BirdList.Add(Bird);
65         }
66     }
```

Figure 373

This section of the code (In the UBird class called, from the MoveBirds class) updates the transform for the mesh to rotate it based on the birds direction.

```
8
9     class UBird
10    {
11        public:
12            FVector Position;
13            FVector Direction;
14            FVector Waypoint;
15            float Speed;
16            USkeletalMeshComponent* BirdMesh = nullptr;
17
18        void UpdateTransform()
19        {
20            if (!BirdMesh)          // Make sure we have a mesh instance before updating transform
21                return;
22
23            FTransform Trans;
24            Trans.SetLocation(Position);
25
26            FRotator Rotator = Direction.Rotation();
27            Trans.SetRotation(FQuat(Rotator));
28
29            BirdMesh->SetWorldTransform(Trans);
30        }
31    };
```

Figure 374

Assigning mesh for birds in the unreal details panel.

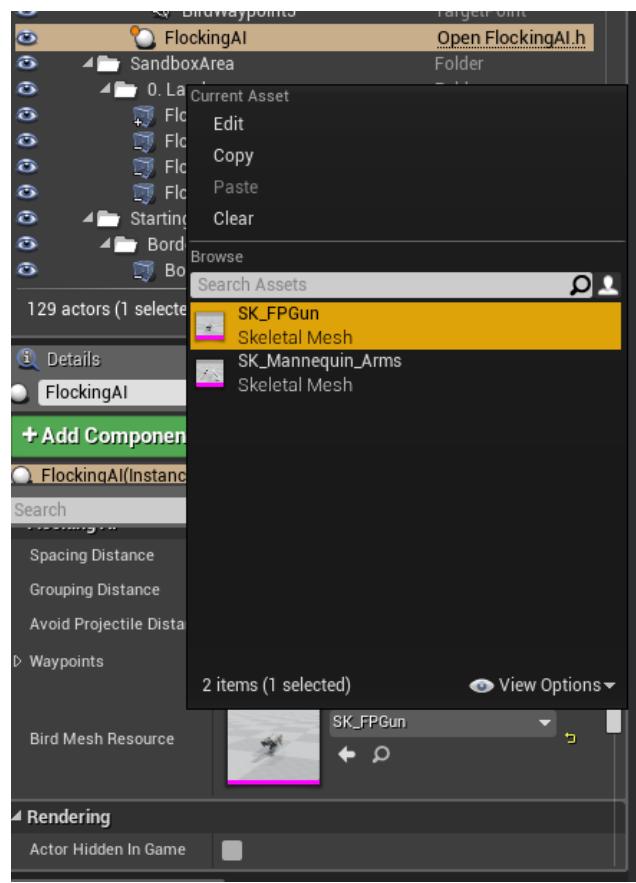


Figure 375

Now the birds have a mesh assigned to them (The gun is obviously temporary).

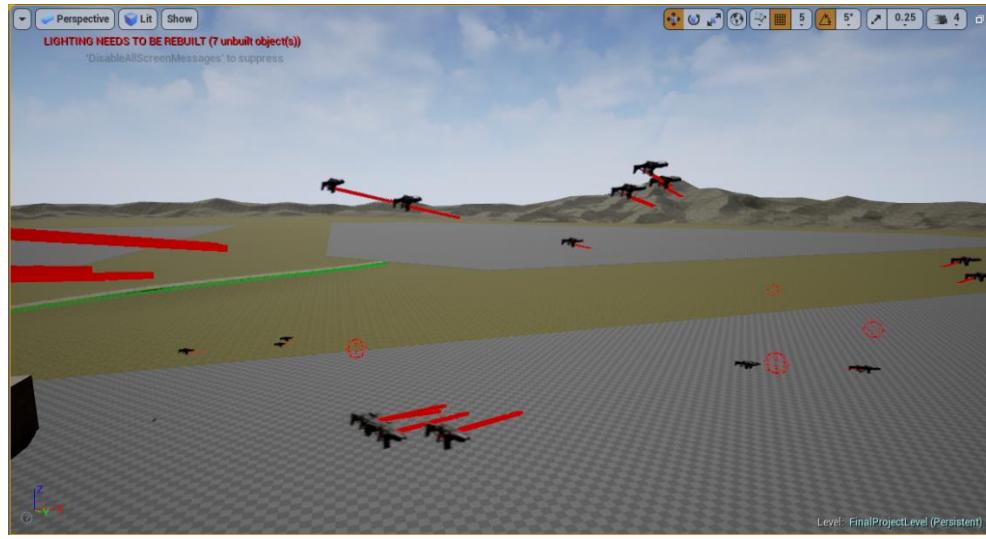


Figure 376

Added a new bird mesh to replace temp gun mesh.

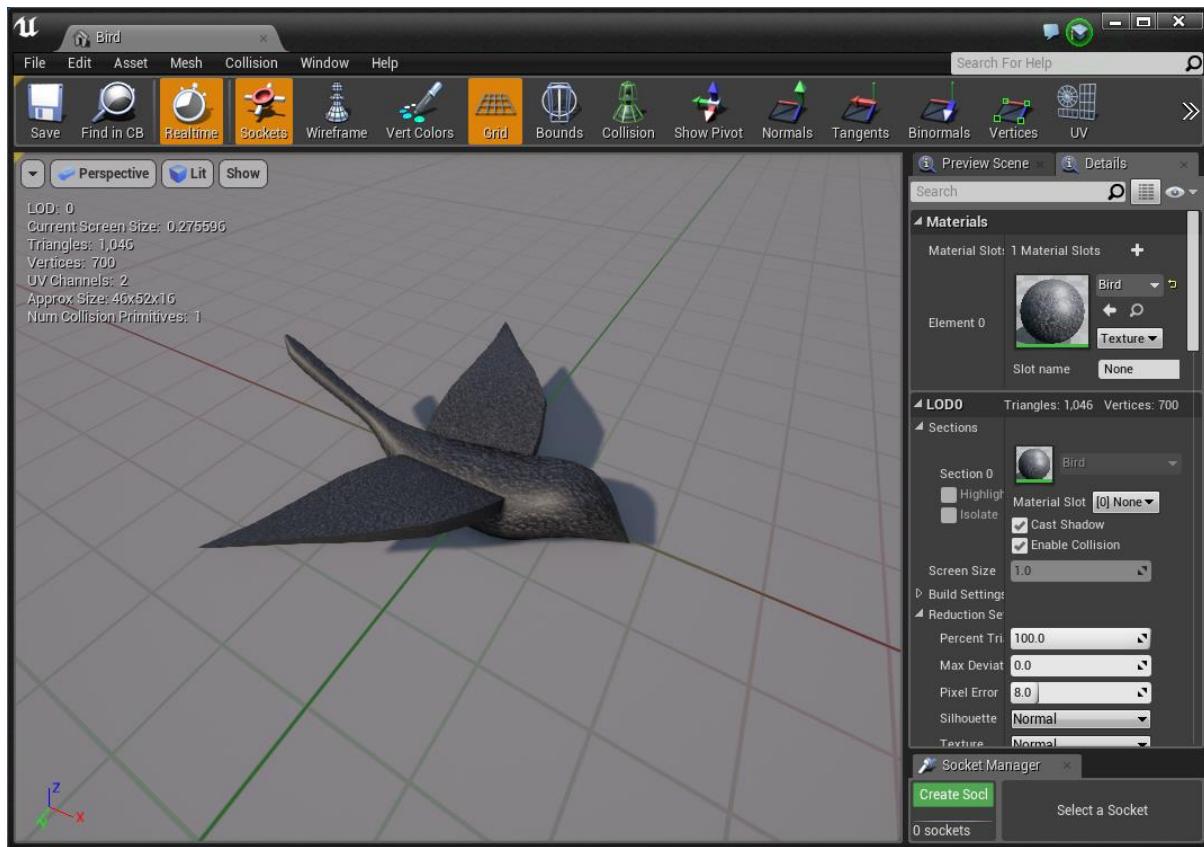


Figure 377

New birds now added to flocking AI and in scene.



Figure 378

Added Podium ready for flocking AI Demo.

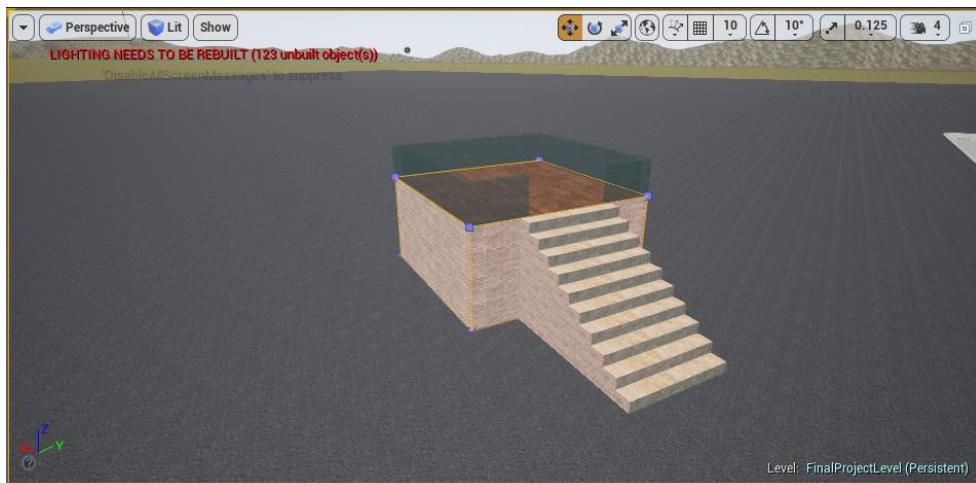


Figure 379

Added a TextContainer component to the FlockingAI Actor and added text to it ready for demo.

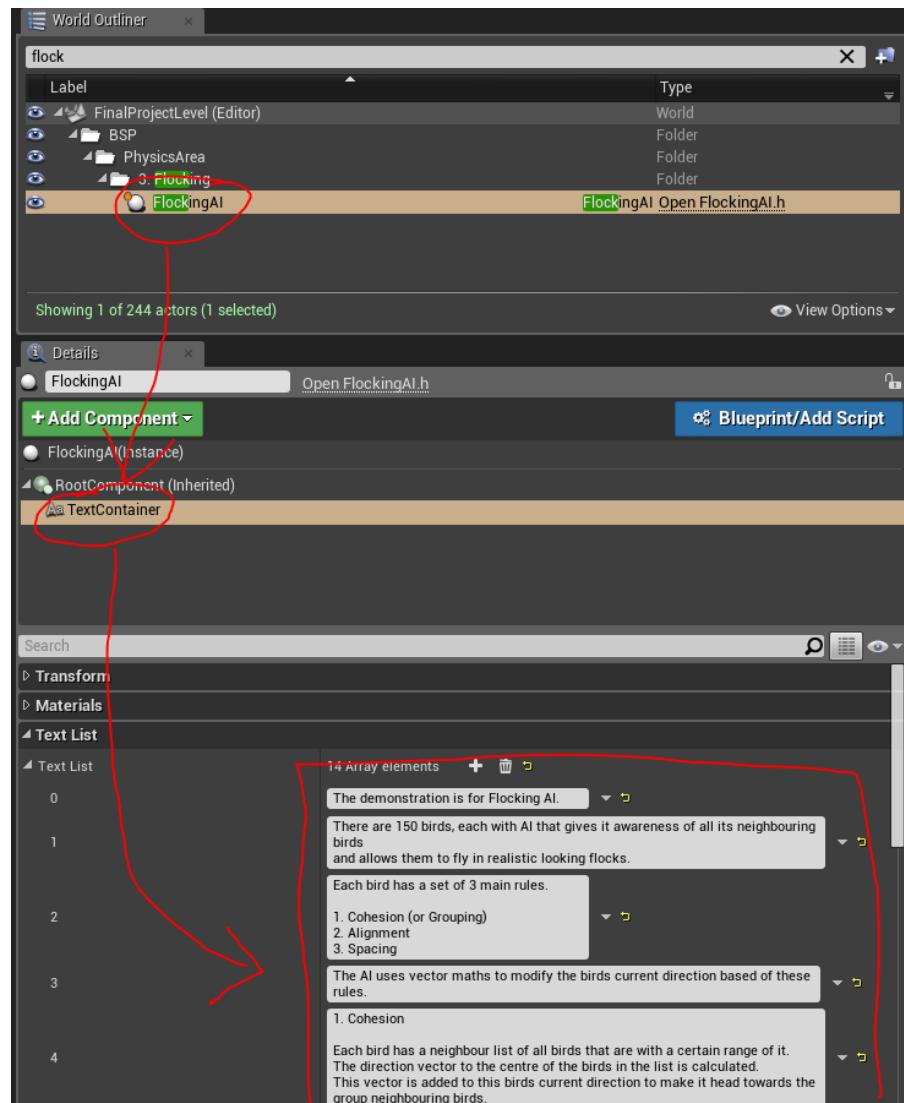


Figure 380

Added TextContainer pointer, TextContainerIndex (int) and UpdateSubTiles function to FlockingAI header file.

```

53
54     UPROPERTY(EditAnywhere)
55         float GroupingDistance = 200.0f;
56
57     UPROPERTY(EditAnywhere)
58         float AvoidProjectileDistance = 1200.0f;
59
60     UPROPERTY(EditAnywhere)
61         TArray<ATargetPoint*> Waypoints;
62
63     UPROPERTY(EditAnywhere)
64         UStaticMesh* BirdMeshResource = nullptr;
65
66     UPROPERTY()
67         UTextContainerComponent* TextContainer = nullptr;
68
69     UPROPERTY(EditAnywhere, Interp)
70         float TextContainerIndex = -1; // -1 default = No Text
71
72 private:
73     UPrimitiveComponent* LastAddedComponent = nullptr;
74
75 private:
76     void MoveBirds(float DeltaTime);
77
78     void ShowWaypoints();
79
80     FVector PickNewWaypoint(FVector OldWaypoint);
81
82     TArray<UBird*> GetNeighbours(UBird& Bird);
83
84     FVector GetDirectionToGroup(UBird& CurrentBird, TArray<UBird*>& Neighbours);
85
86     FVector GetDirectionAwayFromNeighbour(UBird& CurrentBird, TArray<UBird*>& Neighbours);
87     FVector GetAlignmentDirection(UBird& CurrentBird, TArray<UBird*>& Neighbours);
88
89     FVector GetProjectileAvoidDirection(UBird& CurrentBird);
90     void UpdateSubTitleText();
91
92 };
93

```

Figure 381

Added code the BeginPlay to find the TextComponent in the FlockingAI actor and fill in TextComponent pointer with it. Also Initialises TextContainerIndex to -1 (no text).

```

70
71     // Get a list of all UTextContainerComponents
72     TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
73
74     if (TextComponents.Num() > 0)
75         TextContainer = Cast<UTextContainerComponent>(TextComponents[0]);
76
77     TextContainerIndex = -1;
78
79 }

```

Figure 382

Added UpdateSubTitles function to Flocking AI to update HUD subtitle box based on TextContainerIndex.

```
278     // ****
279     // UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
280     // ****
281     void AFlockingAI::UpdateSubTitleText()
282     {
283         if (TextContainer && TextContainerIndex >= 0)
284         {
285             TextContainer->SetTextFromList(-1);           // Turn off 3d text
286
287             FString Temp = TextContainer->GetTextFromList(TextContainerIndex);
288
289             // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
290             while (true)
291             {
292                 int32 Index;
293                 if (!Temp.FindChar( TCHAR('\r'), Index ) )
294                     break;
295                 Temp.RemoveAt( Index, 1 );
296             }
297             AFinalProjectRyanTHUD::SetString( Temp );
298         }
299     }
300 }
```

Figure 383

Added call to said function in Tick.

```
80     // ****
81     // Tick - Called every frame
82     // ****
83     void AFlockingAI::Tick( float DeltaTime )
84     {
85         SCOPE_CYCLE_COUNTER(STAT_FlockingTick);
86
87         Super::Tick( DeltaTime );
88
89         MoveBirds(DeltaTime);
90
91         ShowWaypoints();
92
93         UpdateSubTitleText(); // Circled in red
94     }
95 }
```

Figure 384

## Added Level Sequence triggering code to Flocking (as per VerletSim & Collision Demo)

```

71     UPROPERTY(EditAnywhere, Interp, Category = "Flocking")
72         float TextContainerIndex = -1; // -1 default = No Text
73
74     UPROPERTY(EditAnywhere, Category = "Flocking")
75         ULevelSequence* LevelSequence = nullptr;
76
77     UPROPERTY()
78         ULevelSequencePlayer* SequencePlayer = nullptr;
79
80 private:
81     UPrimitiveComponent* LastAddedComponent = nullptr;
82
83     UPROPERTY(EditAnywhere, Category = "Flocking")
84         UBoxComponent* TriggerVolume = nullptr;
85
86 private:
87     void MoveBirds(float DeltaTime);
88
89     void ShowWaypoints();
90
91     FVector PickNewWaypoint(FVector OldWaypoint);
92
93     TArray<UBird*> GetNeighbours(UBird& Bird);
94
95     FVector GetDirectionToGroup(UBird& CurrentBird, TArray<UBird*>& Neighbours);
96
97     FVector GetDirectionAwayFromNeighbour(UBird& CurrentBird, TArray<UBird*>& Neighbours);
98     FVector GetAlignmentDirection(UBird& CurrentBird, TArray<UBird*>& Neighbours);
99
100    FVector GetProjectileAvoidDirection(UBird& CurrentBird);
101    void UpdateSubTitleText();
102
103    UFUNCTION()
104        void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
105    UFUNCTION()
106        void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
107
108 }

```

```

18 // ****
19 // Constructor
20 // ****
21 AFlockingAI::AFlockingAI()
22 {
23     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
24     PrimaryActorTick.bCanEverTick = true;
25
26     // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
27     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
28
29     // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
30     TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));
31 }
32 // ****

```

```

72
73     // Get a list of all UTextContainerComponents
74     TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
75
76     if ( TextComponents.Num() )
77         TextContainer = Cast<UTextContainerComponent>(TextComponents[0]);
78
79     TextContainerIndex = -1;
80
81     // Setup the trigger volume
82     TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
83     TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
84
85     TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AFlockingAI::EnterVolume);
86     TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AFlockingAI::ExitVolume);
87
88 }

```

```

310 // ****
311 // EnterVolume - Called when component enters collision volume
312 // ****
313 void AFlockingAI::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
314 {
315     // Only allowed if overlapped by any kind of character
316     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
317         return;
318
319     if ( LevelSequence )
320     {
321         FLevelSequencePlaybackSettings Settings;
322
323         Settings.LoopCount = 0;
324
325         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
326         SequencePlayer->Play();
327     }
328 }
329
330 // ****
331 // ExitVolume - Called when component exits collision volume
332 // ****
333 void AFlockingAI::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
334 {
335     // Only allowed if overlapped by any kind of character
336     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
337         return;
338
339     if ( SequencePlayer )
340     {
341         SequencePlayer->Stop();
342         SequencePlayer = nullptr;
343     }
344     TextContainerIndex = -1;
345     AFinalProjectRyanTHUD::SetString();
346 }
347
348

```

Figure 385 – Level Sequence Code (4 Images)

Created empty level sequence ready for scripting.

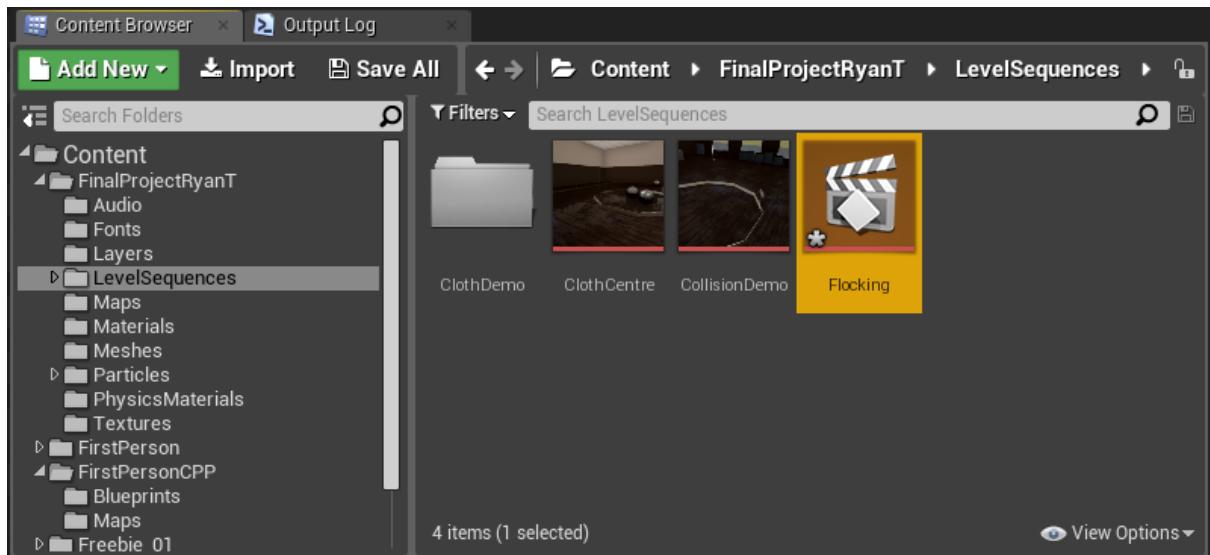


Figure 386

Created a camera in scene ready for scripting.

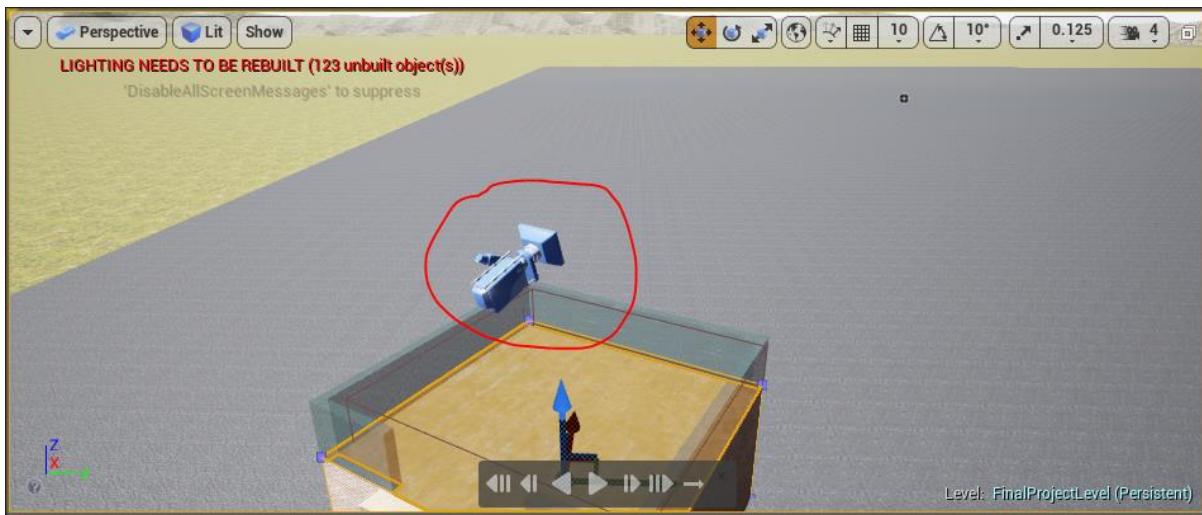


Figure 387

Attached the level sequence to the flocking AI Actor.

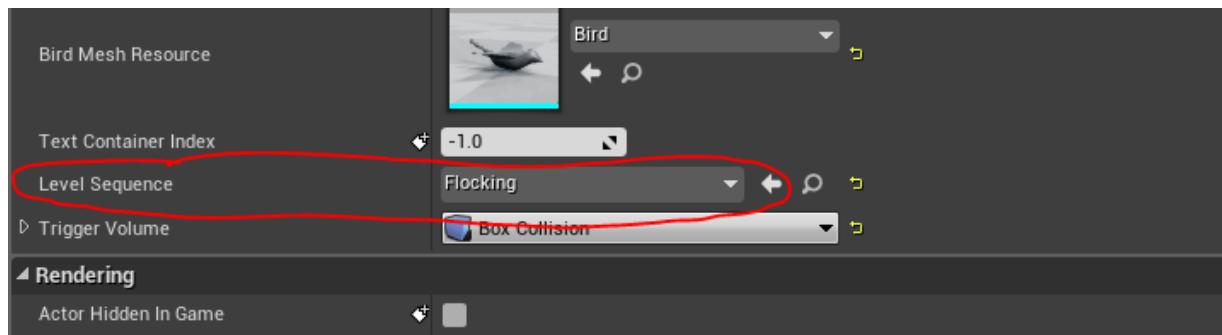


Figure 388

Setup the basic tracks in level sequence. Camera cut and TextContainerIndex. Increments through all text.

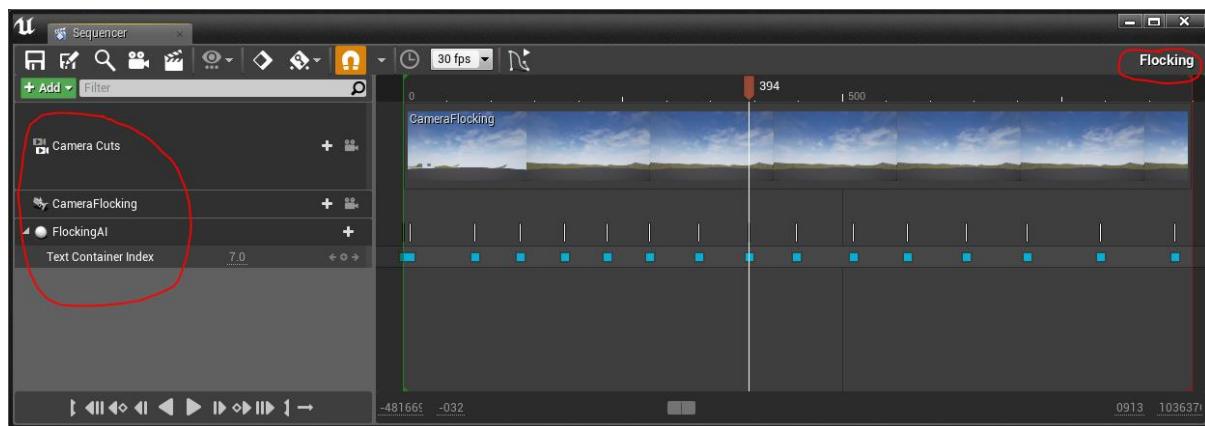


Figure 389

Added flags to flocking AI actor to enable and disable rules to header file.

```
73
74     UPROPERTY(EditAnywhere, Category = "Flocking")
75     ULevelSequence* LevelSequence = nullptr;
76
77     UPROPERTY()
78     ULevelSequencePlayer* SequencePlayer = nullptr;
79
80     UPROPERTY(EditAnywhere, Category = "Flocking|Flags")
81     bool bEnableGrouping = true;
82
83     UPROPERTY(EditAnywhere, Category = "Flocking|Flags")
84     bool bEnableSpacing = true;
85
86     UPROPERTY(EditAnywhere, Category = "Flocking|Flags")
87     bool bEnableAlignment = true;
88
89 private:
90     UPrimitiveComponent* LastAddedComponent = nullptr;
91
92     UPROPERTY(EditAnywhere, Category = "Flocking")
93     UBoxComponent* TriggerVolume = nullptr;
```

Figure 390

Changed MoveBirds function to use said flags.

```
105 // ****
106 // MoveBirds - Moves all birds in BirdList
107 // ****
108 void AFlockingAI::MoveBirds(float DeltaTime)
109 {
110     for (int i = 0; i < BirdList.Num(); i++)
111     {
112         UBird& CurrentBird = BirdList[i];
113
114         TArray<UBird*> Neighbours = GetNeighbours(CurrentBird);
115
116         FVector DirectionToGroup = FVector(0,0,0); // Initially zero vectors
117         FVector DirectionAwayFromNeighbour = FVector(0,0,0);
118         FVector AlignmentDirection = FVector(0,0,0);
119
120         if ( bEnableGrouping )
121             DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);
122
123         if ( bEnableSpacing )
124             DirectionAwayFromNeighbour = GetDirectionAwayFromNeighbour(CurrentBird, Neighbours);
125
126         if ( bEnableAlignment )
127             AlignmentDirection = GetAlignmentDirection(CurrentBird, Neighbours);
128
129         FVector ProjectileAvoid = GetProjectileAvoidDirection(CurrentBird);
130
131         FVector ToWaypoint = CurrentBird.Waypoint - CurrentBird.Position;
132         ToWaypoint.Normalize();
133 }
```

Figure 391

Had to change UBirds class into an Unreal structure type to allow for garbage collection on the BirdMesh pointer. Unfortunately this also required renaming it to FBird otherwise Unreal gives error about Naming Convention.

```
12  USTRUCT()
13  struct FBird
14  {
15      GENERATED_BODY()
16
17  public:
18
19      ~FBird()
20      {
21          if (BirdMesh)
22              BirdMesh->UnregisterComponent();
23          BirdMesh = nullptr;
24      }
25
26      FVector Position;
27      FVector Direction;
28      FVector Waypoint;
29      float Speed;
30
31  UPROPERTY()
32      UStaticMeshComponent* BirdMesh = nullptr;
33
34  void UpdateTransform()
35  {
36      if (!BirdMesh) // Make sure we have a mesh instance before updating transform
37          return;
38
39      FTransform Trans;
40      Trans.SetLocation(Position);
41
42      FRotator Rotator = Direction.Rotation();
43      Trans.SetRotation(FQuat(Rotator));
44
45      BirdMesh->SetWorldTransform(Trans);
46  }
47 };
```

Figure 392

Moved code to create birds from BeginPlay into new func CreateBirds(). Also created DestroyBirds().

```

332 // ****
333 // CreateBirds - Creates all birds for the demonstration
334 // ****
335 void AFlockingAI::CreateBirds()
336 {
337     int MaxBirds = 150;//100;
338
339     FBird Bird;
340
341     for (int i = 0; i < MaxBirds; i++)
342     {
343         FVector Offset(FMath::FRandRange(-2000, 2000), FMath::FRandRange(-3000, 3000), FMath::FRandRange(-200, 200));
344
345         Bird.Position = GetActorLocation() + Offset;           // Waypoint location plus rand offset
346         Bird.Direction = FVector(-1, 0, 0);
347         Bird.Waypoint = PickNewWaypoint(Bird.Position);        // Pick new waypoint.
348         Bird.Speed = 400.0f;
349         Bird.Speed = 300.0f;
350
351         // Create a skeletal mesh instance for this bird, using BirdMeshResource as a resource
352         Bird.BirdMesh = NewObject<UStaticMeshComponent>(this, NAME_None);
353
354         Bird.BirdMesh->SetStaticMesh(BirdMeshResource);
355
356         Bird.UpdateTransform();
357         Bird.BirdMesh->RegisterComponent();
358         Bird.BirdMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);    // Don't need collision
359
360         // KeepWorldTransform means that each component is not relative to the flockingAI actor, it has its own worldspace position
361         Bird.BirdMesh->AttachToComponent(LastAddedComponent ? GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);
362
363         LastAddedComponent = Bird.BirdMesh;
364
365         BirdList.Add(Bird);
366     }
367 }
368
369 // ****
370 // DestroyBirds - Destroys all birds
371 // ****
372 void AFlockingAI::DestroyBirds()
373 {
374     BirdList.Empty();
375     LastAddedComponent = nullptr;
376 }
377

```

Figure 393

Called said functions when entering or leaving trigger volume.

```

290 // ****
291 // EnterVolume - Called when component enters collision volume
292 // ****
293 void AFlockingAI::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromPhysics)
294 {
295     // Only allowed if overlapped by any kind of character
296     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
297         return;
298
299     CreateBirds();
300
301     if ( LevelSequence )
302     {
303         FLevelSequencePlaybackSettings Settings;
304
305         Settings.LoopCount = 0;
306
307         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
308         SequencePlayer->Play();
309     }
310 }
311
312 // ****
313 // EnterVolume - Called when component exits collision volume
314 // ****
315 void AFlockingAI::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
316 {
317     // Only allowed if overlapped by any kind of character
318     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
319         return;
320
321     if ( SequencePlayer )
322     {
323         SequencePlayer->Stop();
324         SequencePlayer = nullptr;
325     }
326     TextContainerIndex = -1;
327     AFinalProjectRyanTHUD::SetString();
328
329     DestroyBirds();
330 }
331

```

Figure 394

### 35.6 Commented MoveBirds function.

```
83     void AFlockingAI::MoveBirds(float DeltaTime)
84     {
85         for (int i = 0; i < BirdList.Num(); i++)
86         {
87             FBird& CurrentBird = BirdList[i];
88
89             // Get a list of neighbouring birds
90             TArray<FBird*> Neighbours = GetNeighbours(CurrentBird);
91
92             // Accumulate any direction changes needed
93             FVector DirectionToGroup = FVector(0,0,0);           // Initially zero vectors
94             FVector DirectionAwayFromNeighbour = FVector(0,0,0);
95             FVector AlignmentDirection = FVector(0,0,0);
96
97             // Grouping
98             if (bEnableGrouping)
99                 DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);
100
101            // Spacing
102            if (bEnableSpacing)
103                DirectionAwayFromNeighbour = GetDirectionAwayFromNeighbour(CurrentBird, Neighbours);
104
105            // Alignment
106            if (bEnableAlignment)
107                AlignmentDirection = GetAlignmentDirection(CurrentBird, Neighbours);
108
109            // Projectiles
110            FVector ProjectileAvoid = GetProjectileAvoidDirection(CurrentBird);
111
112            // Waypoint
113            FVector ToWaypoint = CurrentBird.Waypoint - CurrentBird.Position;      // The vector from bird to waypoint
114            ToWaypoint.Normalize();
115
116            FVector Dir = ToWaypoint;          // The base direction is head to waypoint
117
118            Dir += AlignmentDirection;        // Add the direction checge for alignment
119            Dir += DirectionToGroup;         // Add the direction checge for grouping
120            Dir += DirectionAwayFromNeighbour; // Add the direction checge for spacing
121            Dir += ProjectileAvoid;         // Add the direction checge for avoiding projectiles
122
123            // Turn the bird
124            CurrentBird.Direction += Dir * 0.00f;
125            CurrentBird.Direction += Dir * 0.04f; // Turn the bird
126
127            CurrentBird.Direction.Normalize();
128
129            // Add direction to current position * it's speed
130            CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;
131
132            if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 100.0f)    // If bird is within 5 metres of waypoint...
133            {
134                CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint);           // Pick new waypoint.
135            }
136
137            CurrentBird.UpdateTransform();
138        }
    }
```

Figure 395

Added UPROPERTY bool to flocking to enable heading debug draw in sequence.

```
97         bool bEnableGrouping = true;
98
99         UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
100        bool bEnableSpacing = true;
101
102        UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
103        bool bEnableAlignment = true;
104
105        UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
106        bool bShowBirdsHeadings = false;
107
108    private:
109        UPrimitiveComponent* LastAddedComponent = nullptr;
110
111        UPROPERTY(EditAnywhere, Category = "Flocking")
112        UBoxComponent* TriggerVolume = nullptr;
113
```

Figure 396

Added code to draw birds required direction and current direction when bool is true.

```
127     // Add direction to current position * it's speed
128     CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;
129
130     if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 100.0f)    // If bird is within 5 metres of waypoint...
131     {
132         CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint);           // Pick new waypoint.
133     }
134
135     CurrentBird.UpdateTransform();
136
137     if ( bShowBirdsHeadings )
138     {
139         DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + CurrentBird.Direction * 60, 20.0f, FColor::Red, false, -1.0f, 0, 2.0f);   // Current direction
140         DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + Dir * 80, 20.0f, FColor::Green, false, -1.0f, 0, 2.0f);    // Desired heading
141     }
142 }
143 }
```

Figure 397

Added tracks to Level Sequence to script the demo.

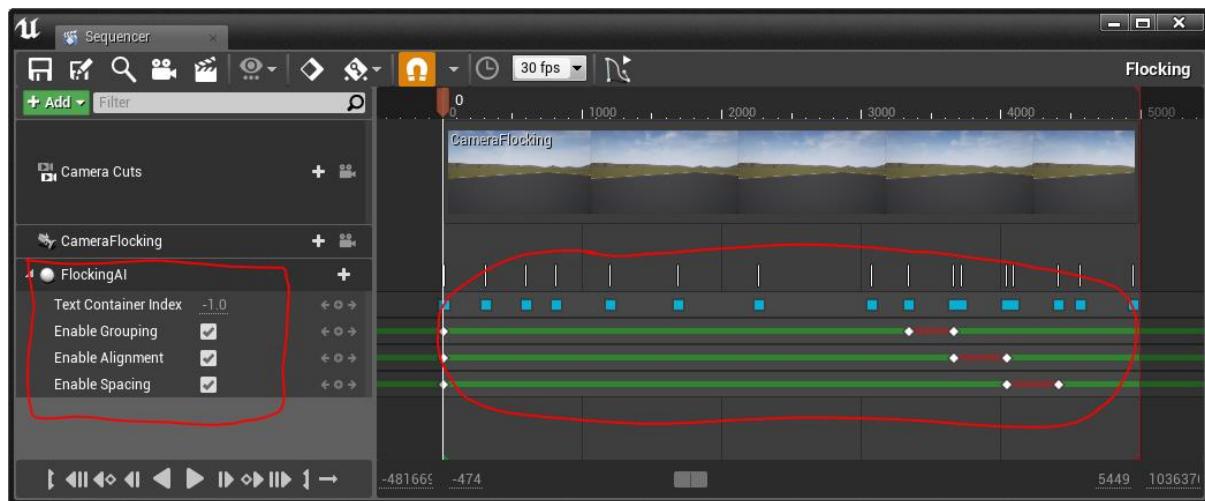
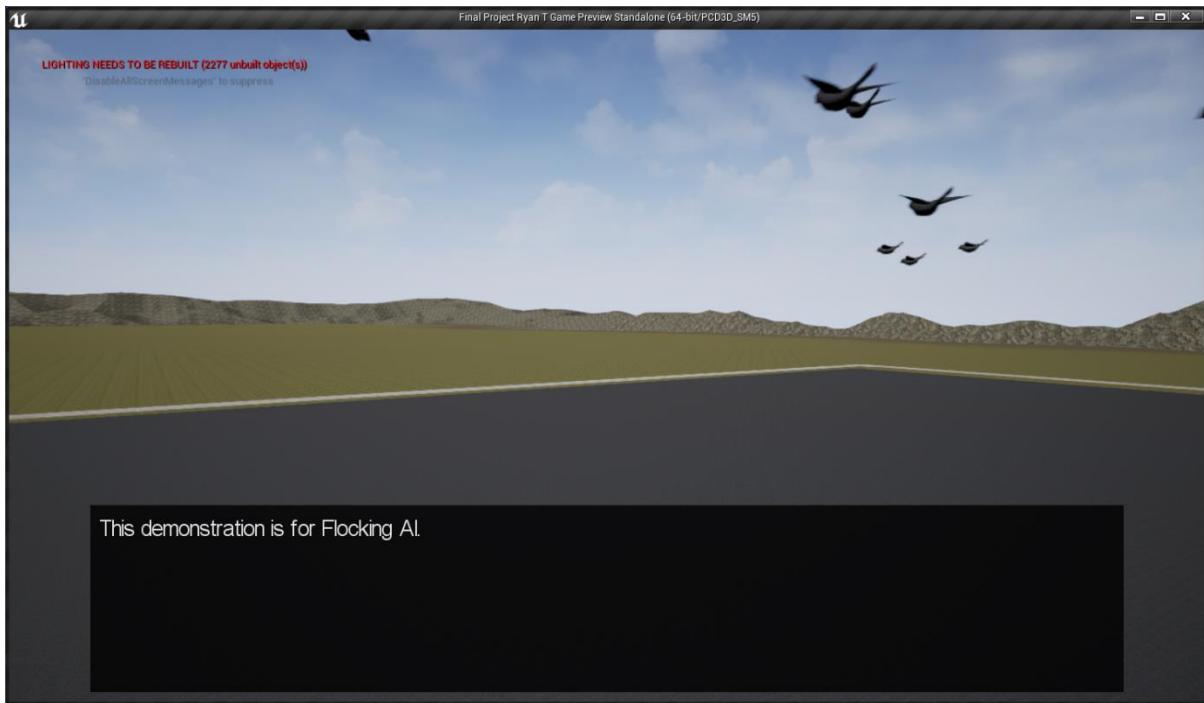
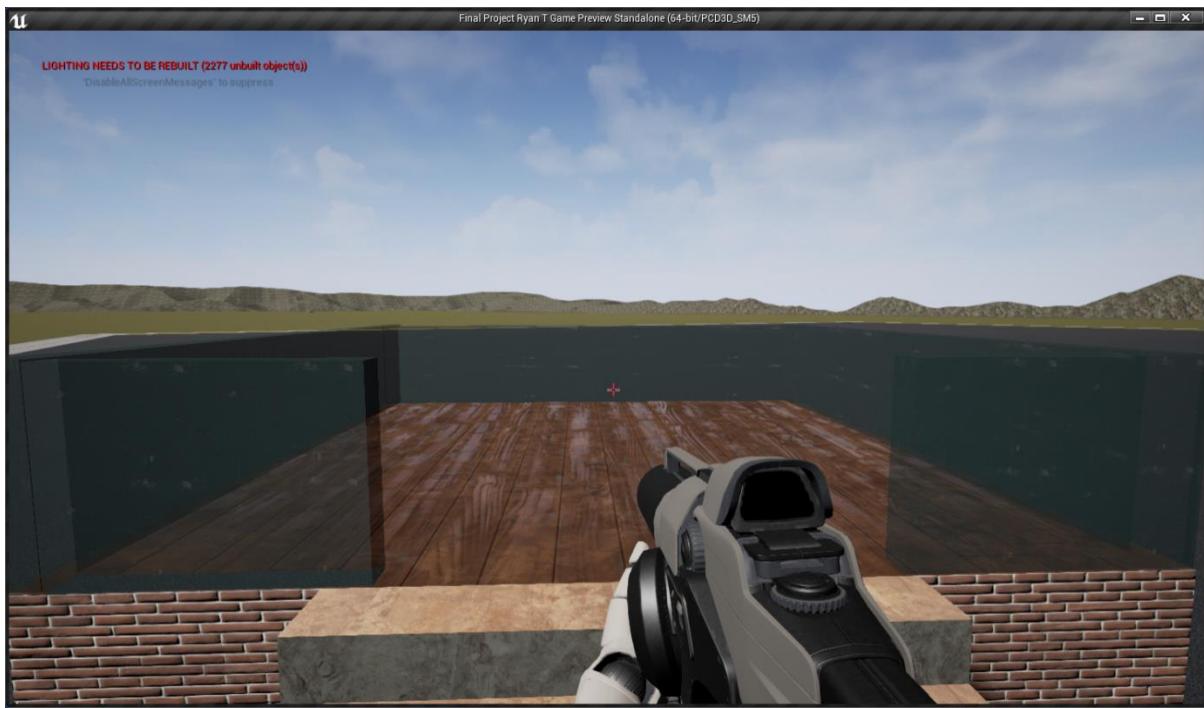
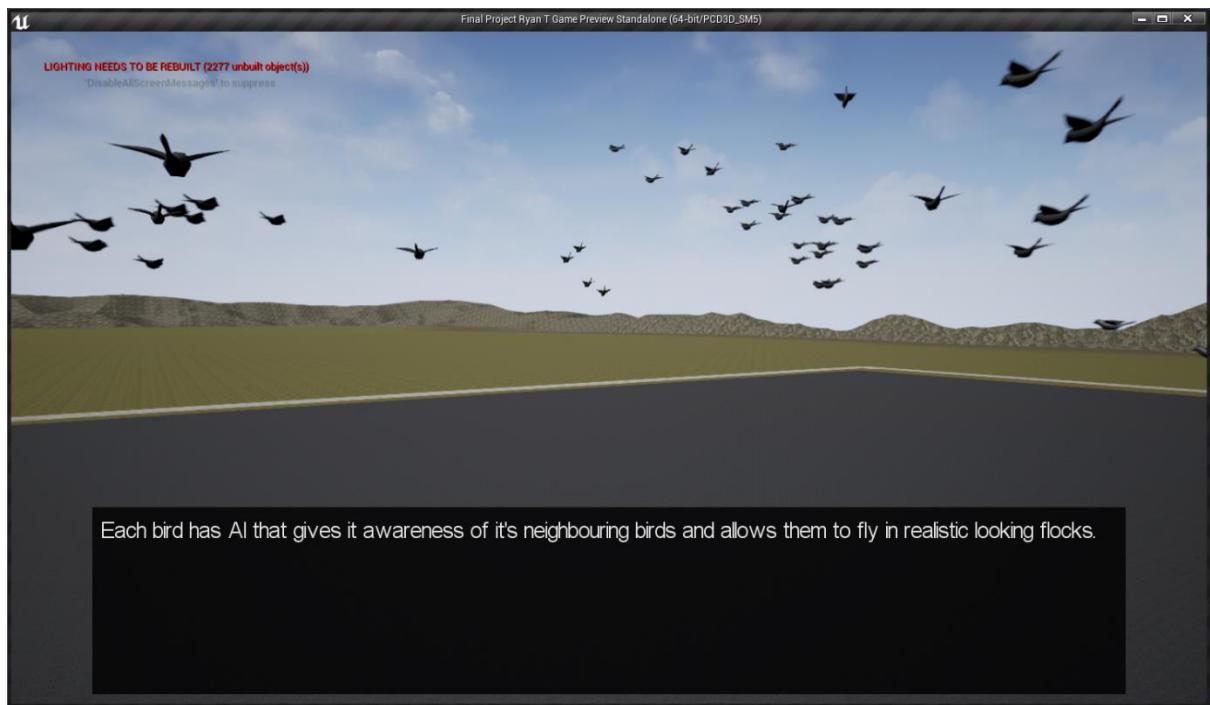


Figure 398

## The completed Demo...









### 2. Alignment

The directions of the neighbours are averaged to get the group direction vector.

This is added to this bird's current direction to make it turn towards the group's general direction.



### 3. Spacing

We go through the neighbours and find any birds that are closer than the required spacing distance.  
For each bird that is, we calculate the vector away from it.

This is weighted by how close the 2 birds are together so the force is greater the closer the birds are.  
This is added to the bird's current direction to move it away.



That is essentially it.

These rules will now be demonstrated.



Grouping OFF





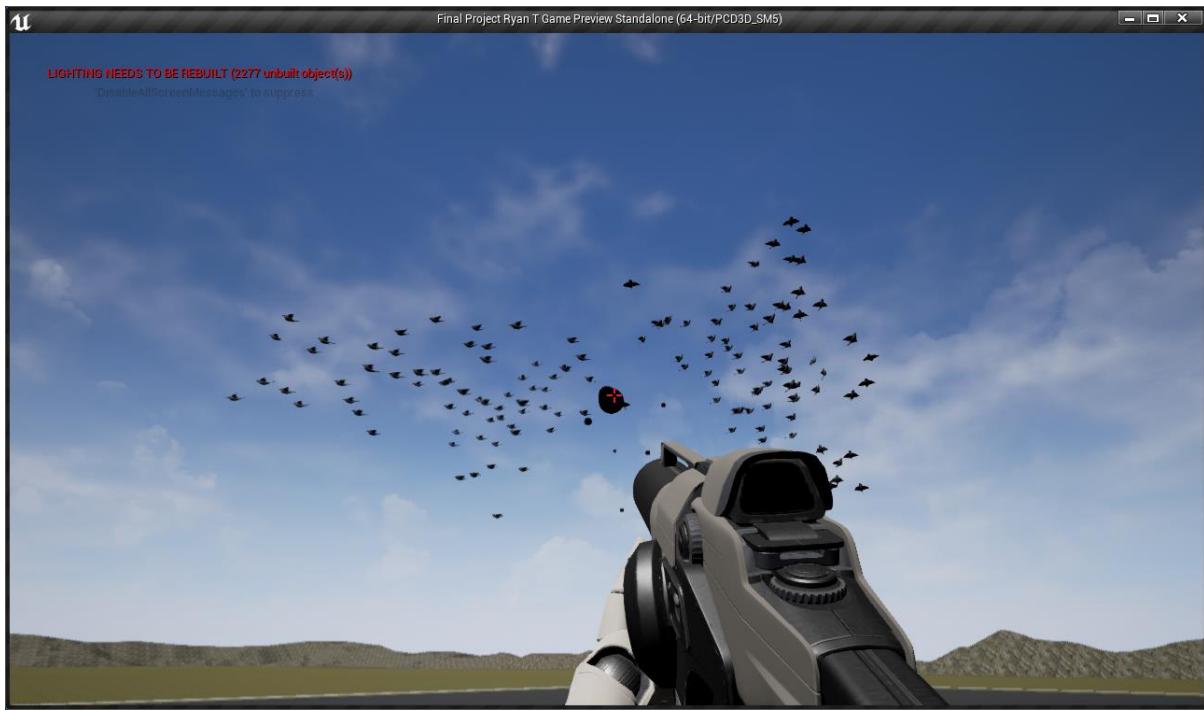


Figure 399 – Flocking Demonstration (16 Images)

## 6.5. Level Changes & Finishing Touches

Selecting physics area, going to scale it down.



Figure 400

Area scaled down to 50x30, and moved into place.



Figure 401

Lighting area also scaled down and moved.

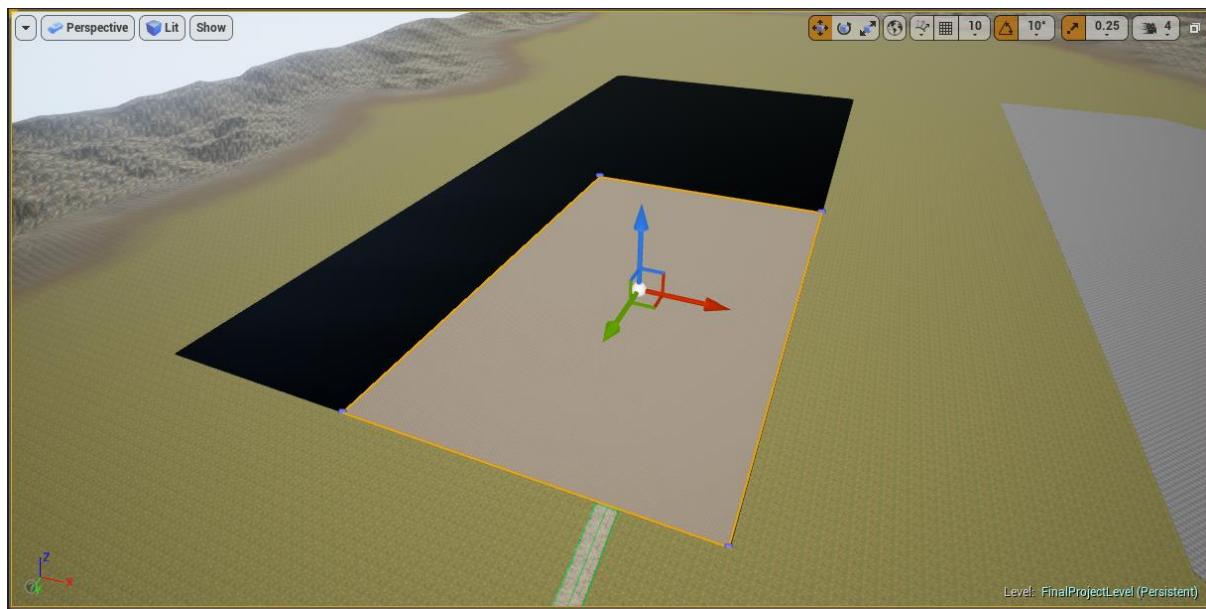
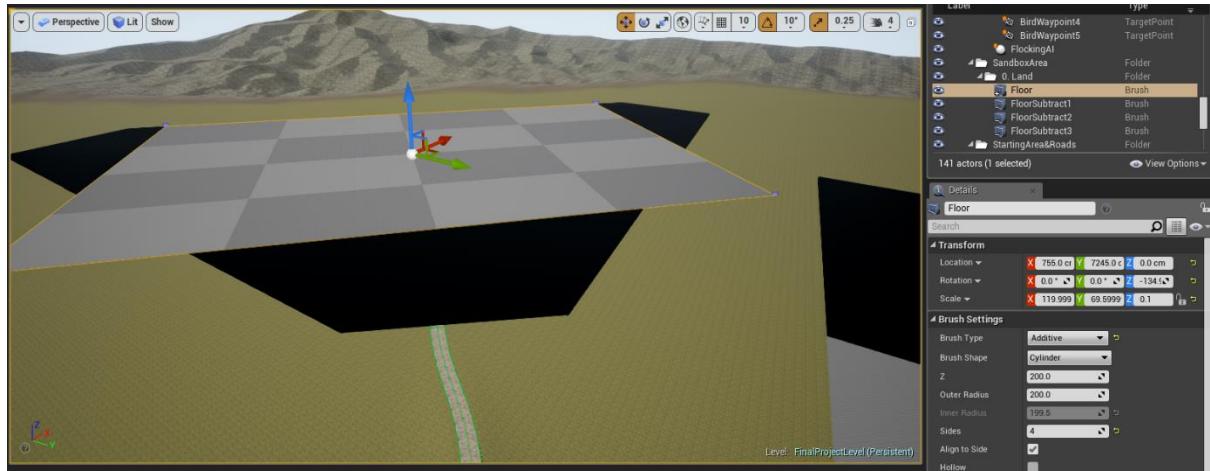
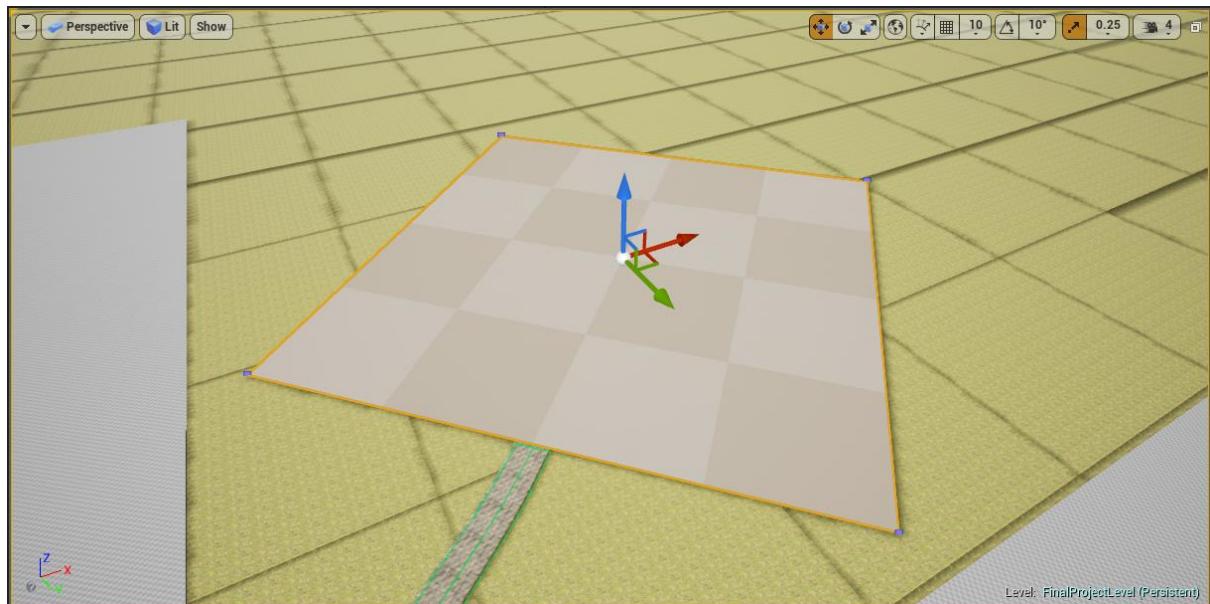


Figure 402

Sandbox area made to have 4 sides now rather than 3.



Sandbox area now scaled and moved.



Paths made smaller, and everything moved in closer.

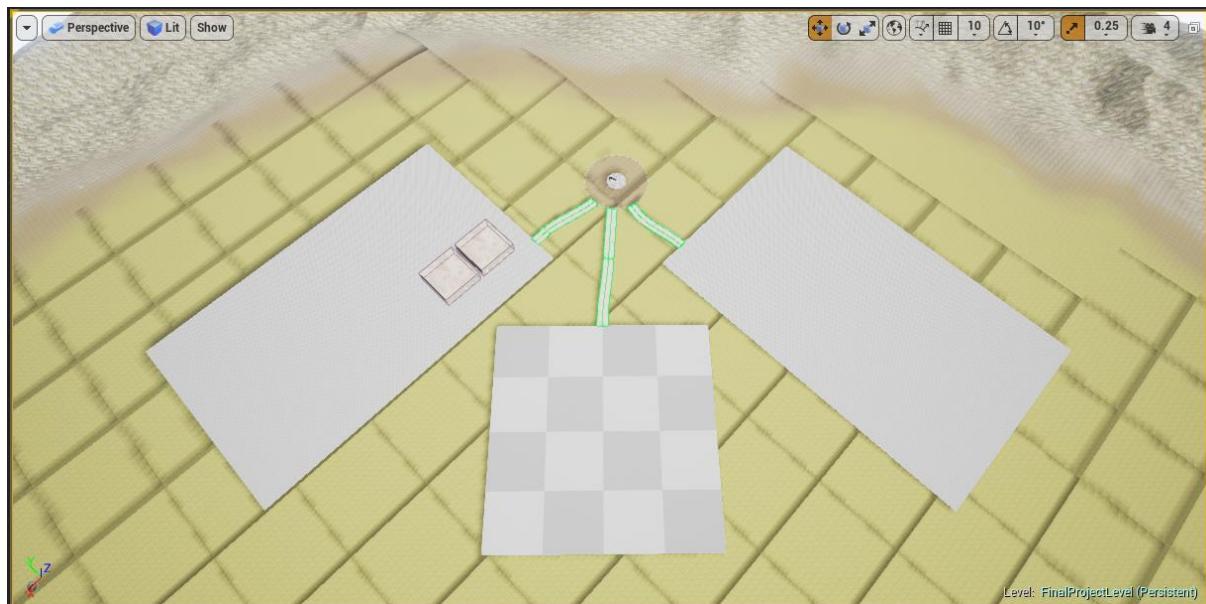


Figure 405

Considering other changes were being made, a new asset pack was downloaded to replace old placeholder buildings.

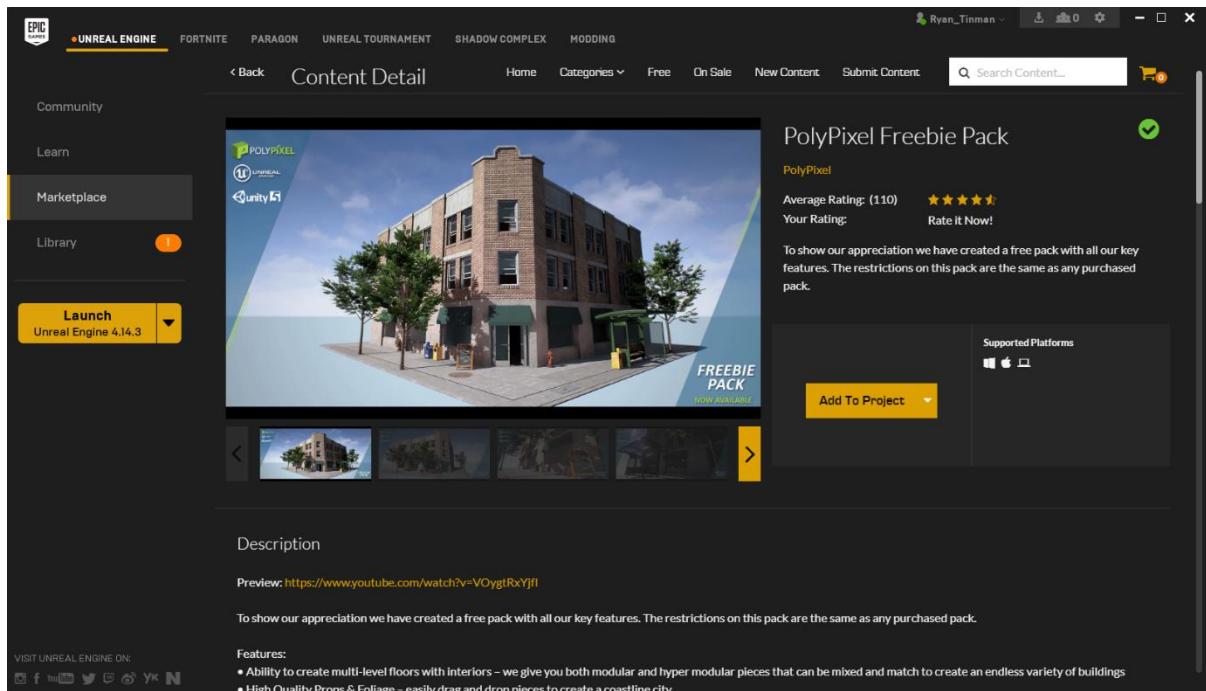


Figure 406

New building mesh has no collision, so a new collision mesh must be made.

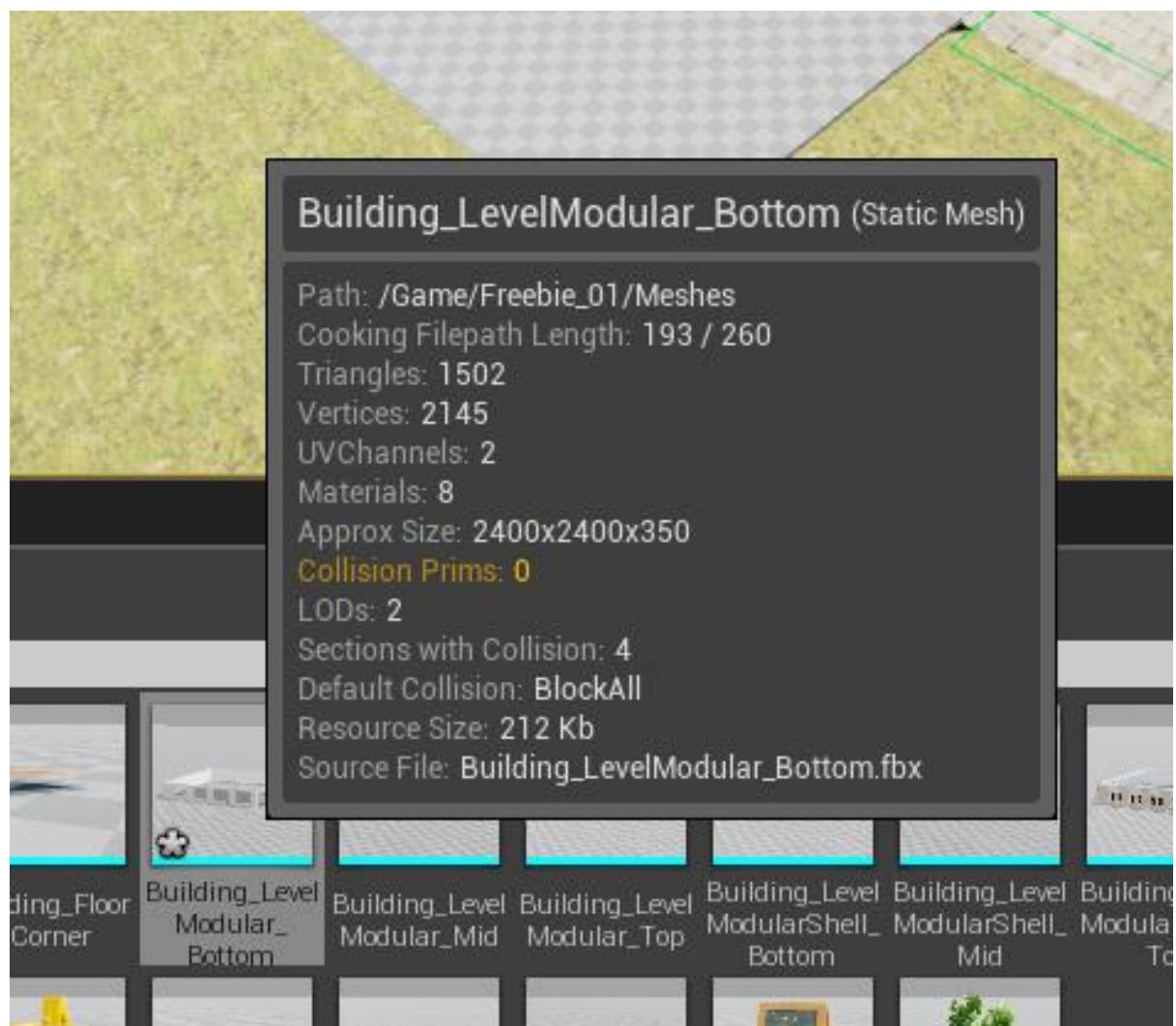


Figure 407

Here is the mesh editor, here collision meshes can be added to objects.

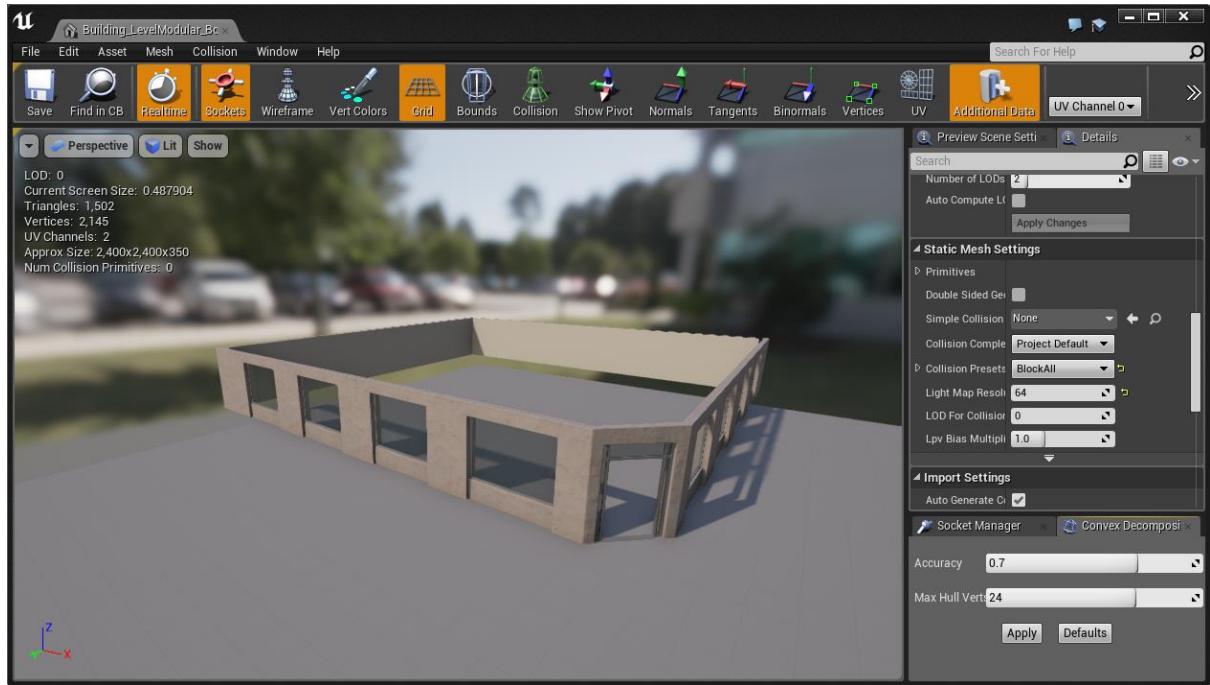


Figure 408

Setting the collision complexity to 'Use complex collision as simple' will use the mesh's polys to make collision.

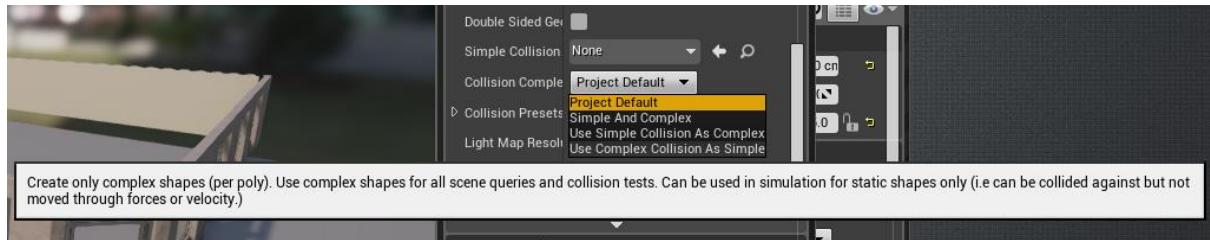


Figure 409

Collision added.

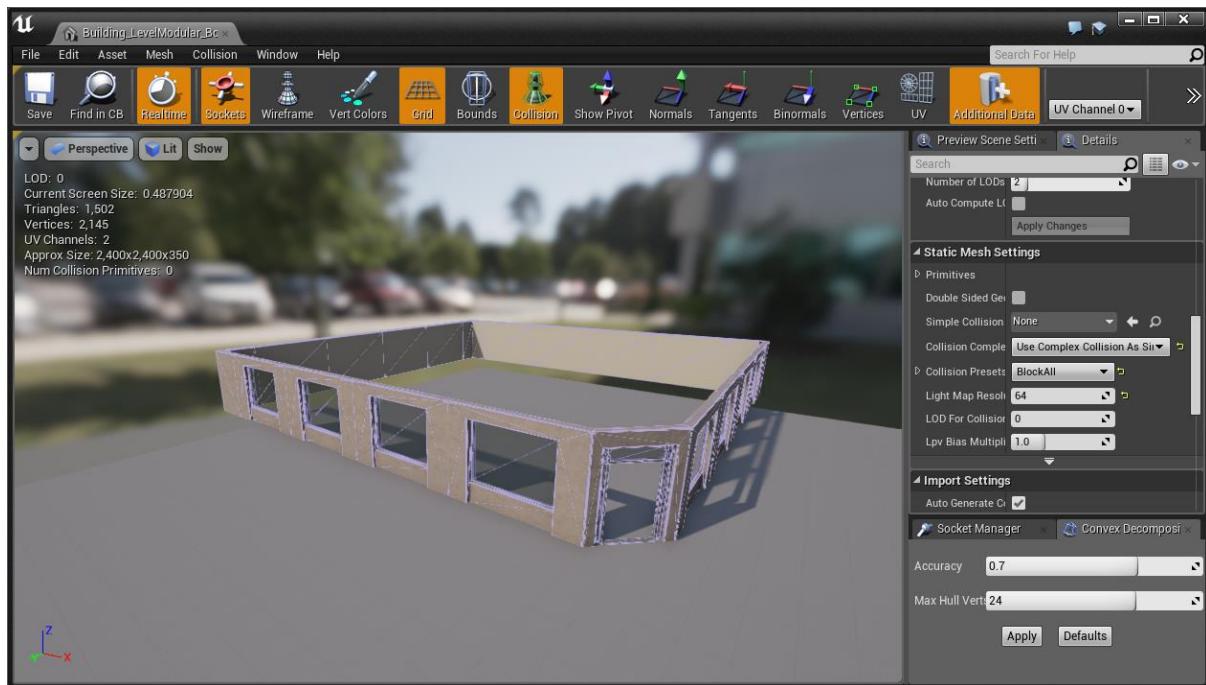


Figure 410

Removed placeholder building and placed ground floor.



Figure 411

Added wood floors.



Figure 412

Added first floor and roof.



Figure 413

Added front doors (will be animated eventually) and path.



Figure 414

Path complete.



Figure 415

Added road texture to floor.



Figure 416

Both buildings replaced.



Figure 417

Added molecule building.



Figure 418

Made lighting buildings.



Figure 419

Made inaccessible buildings for physics area.



Figure 420

Made inaccessible buildings for lighting area.



Figure 421

Made outer path for physics area.



Figure 422

Made outer path for lighting area.



Figure 423

Made outer path for sandbox area.

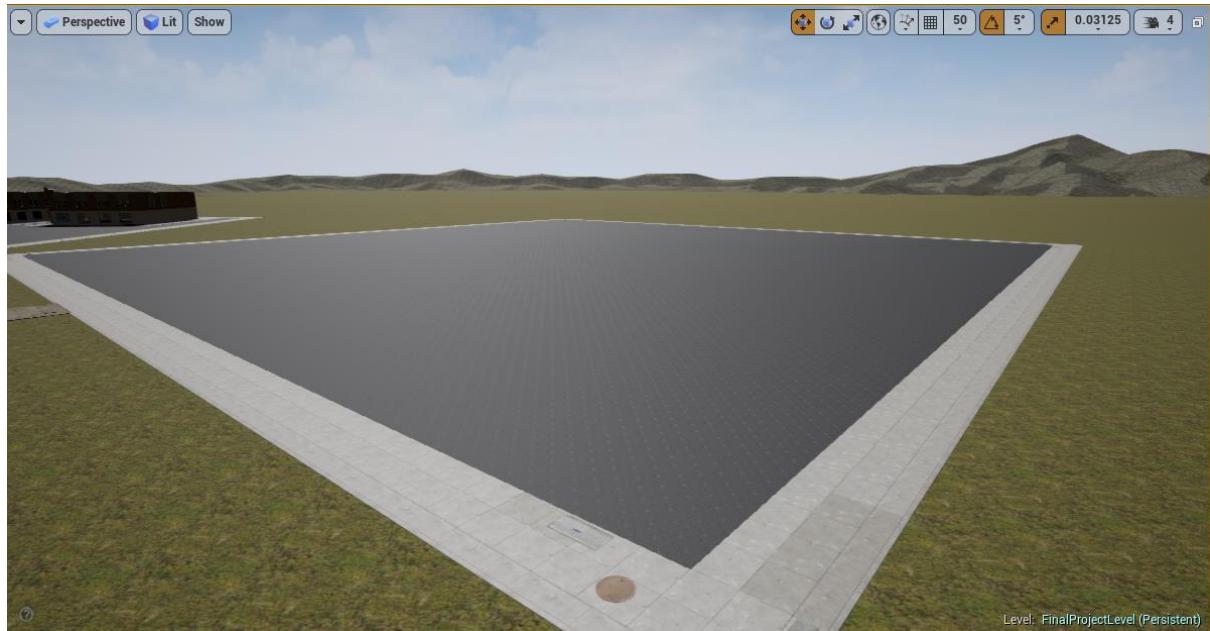


Figure 424

Going to add blocking volumes.

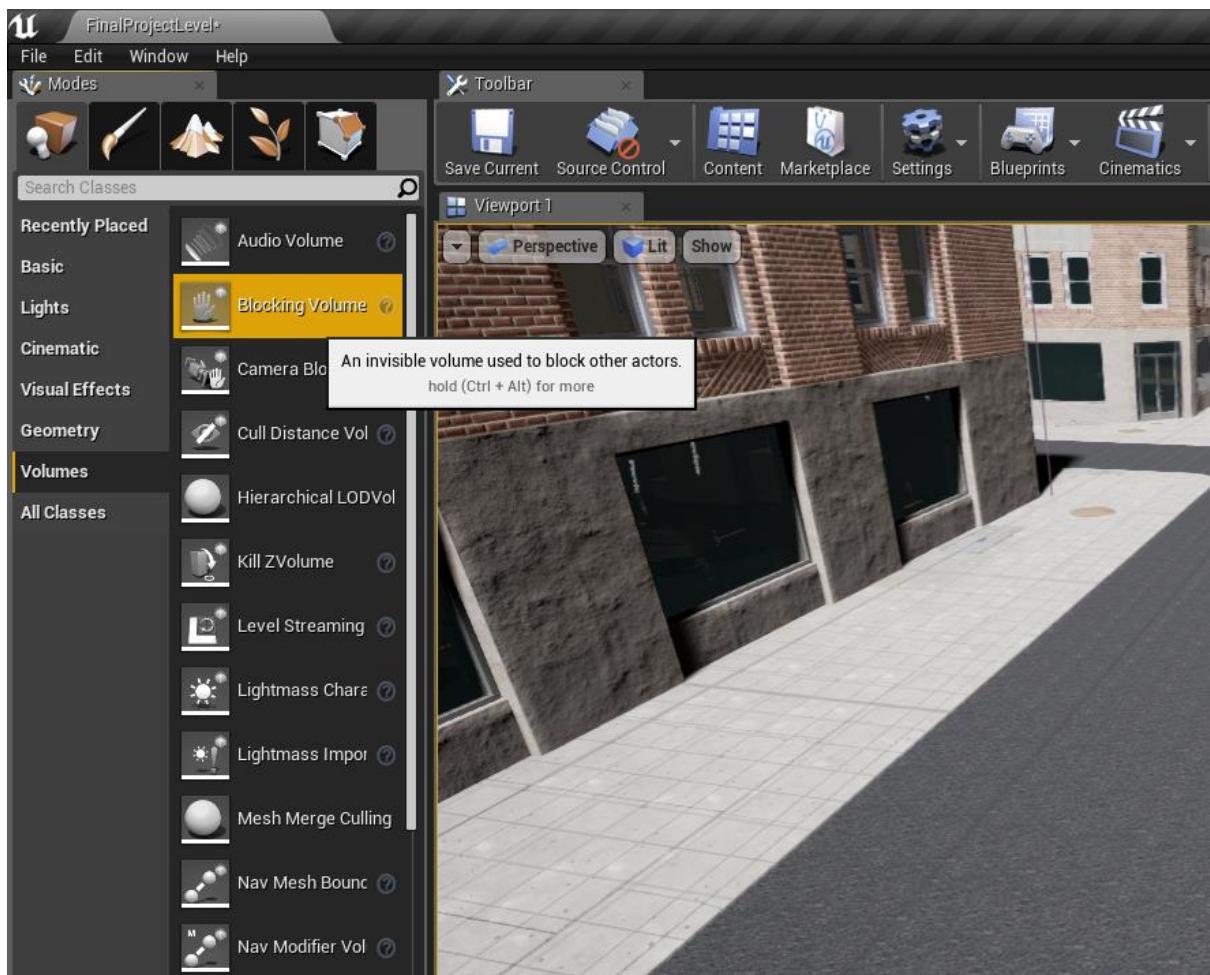


Figure 425

Placed blocking volume.

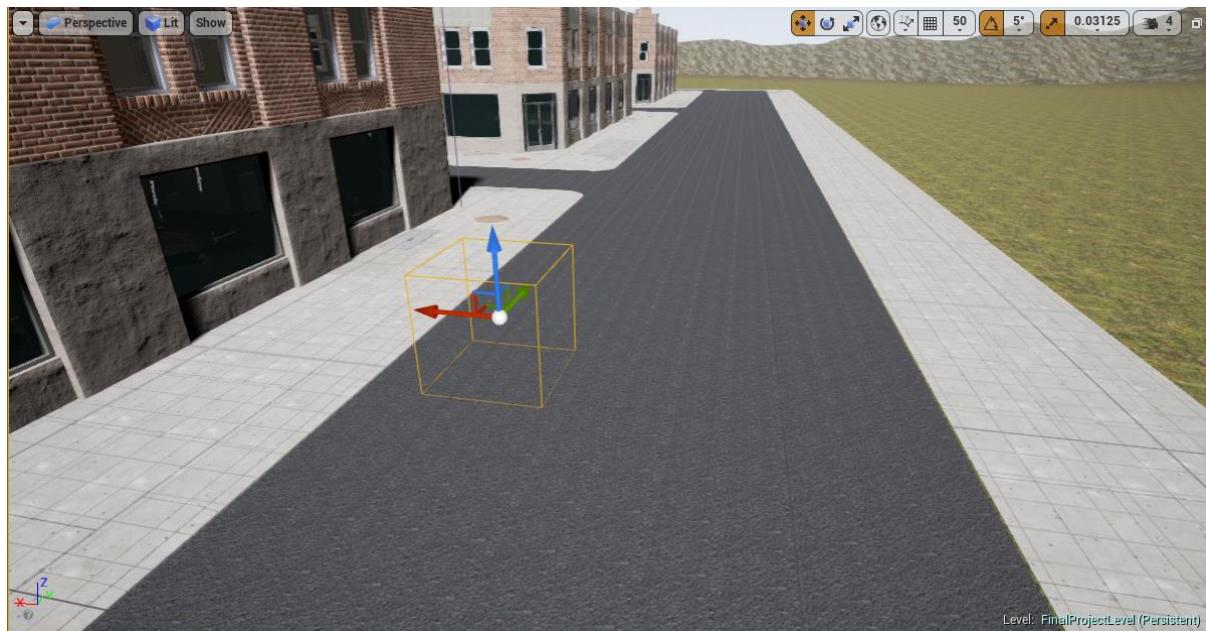


Figure 426

Sized volume.

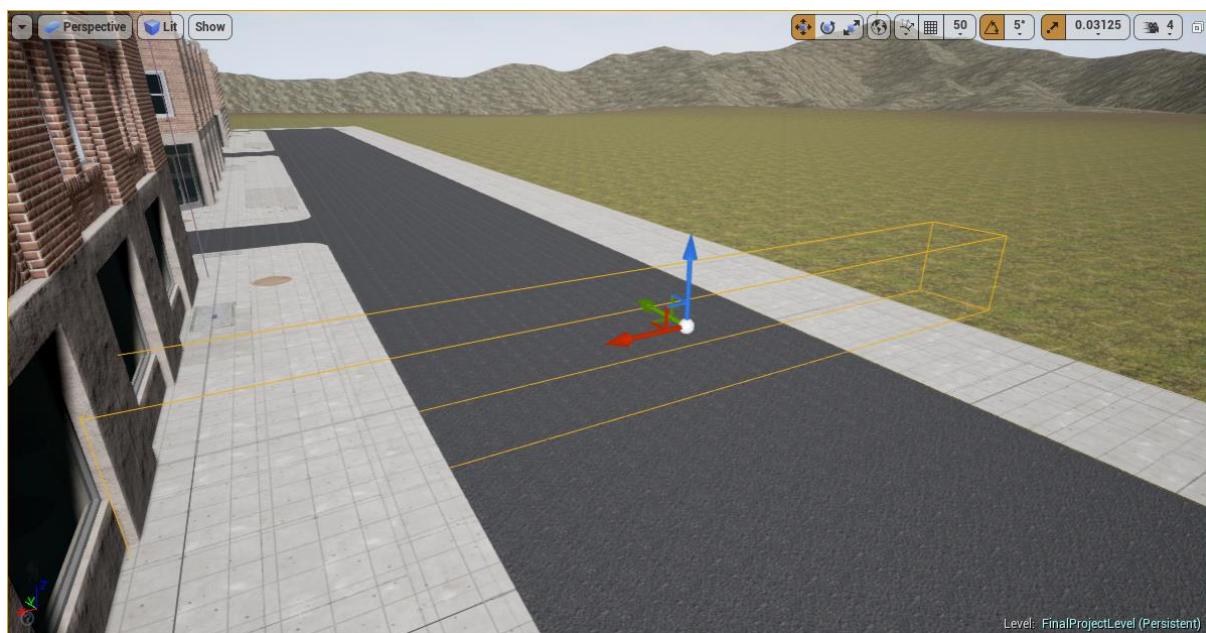


Figure 427

Here are blocking volume settings, they need changing because it stops projectiles, we only want to stop player.

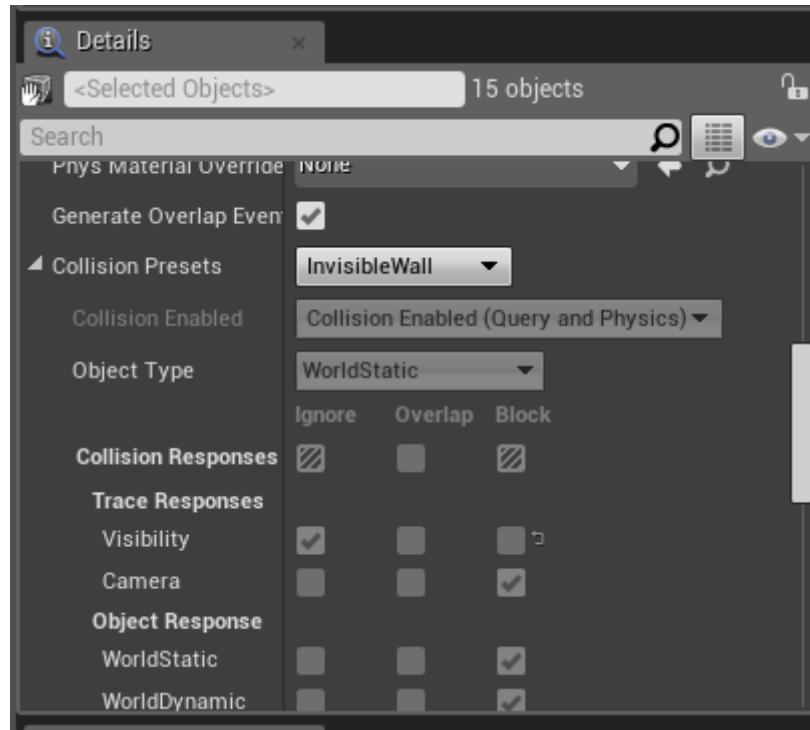


Figure 428

New settings.

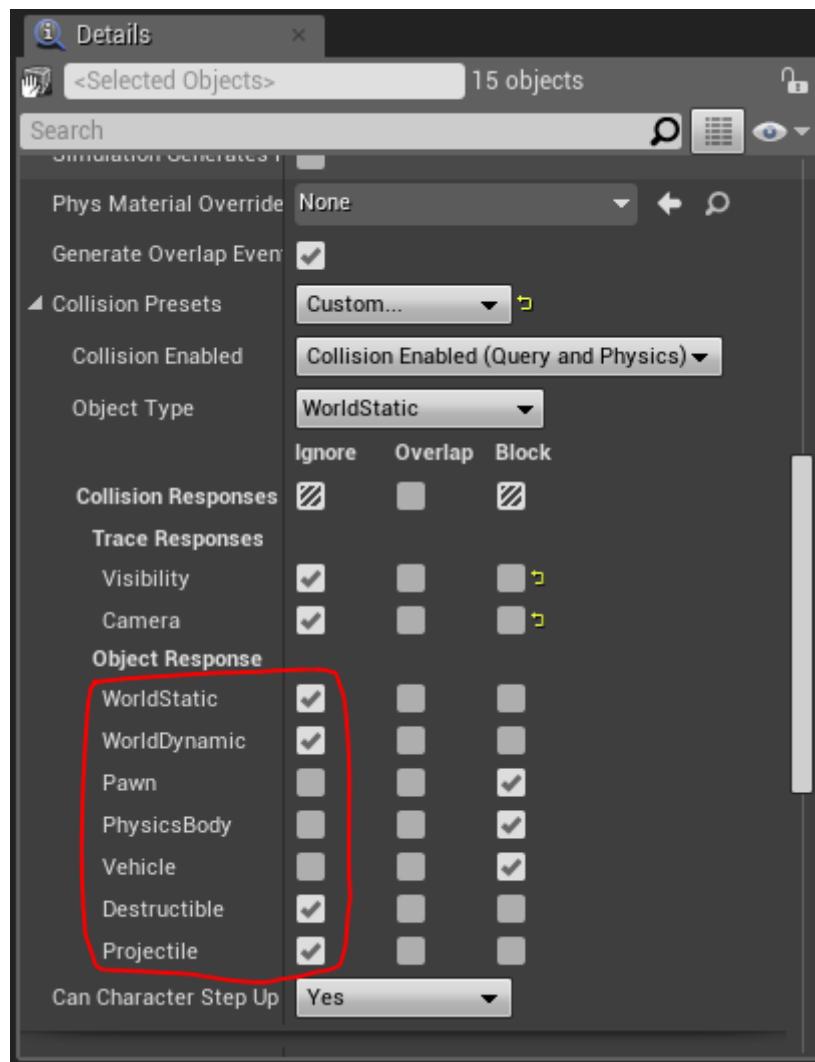


Figure 429

Blocking volumes set up around map.

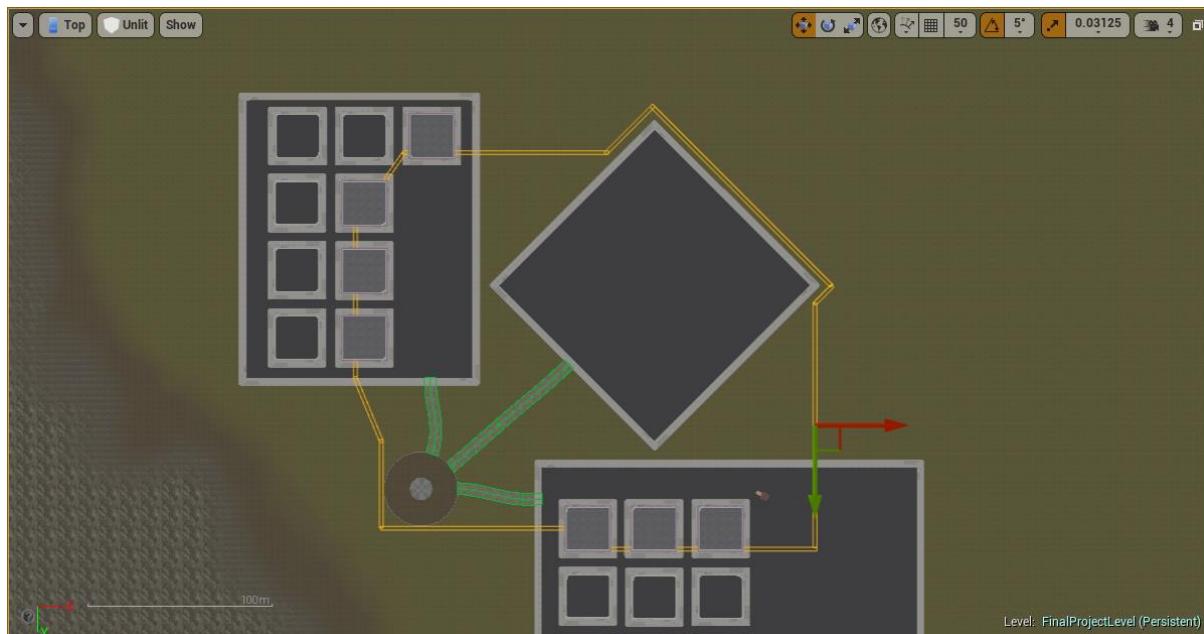
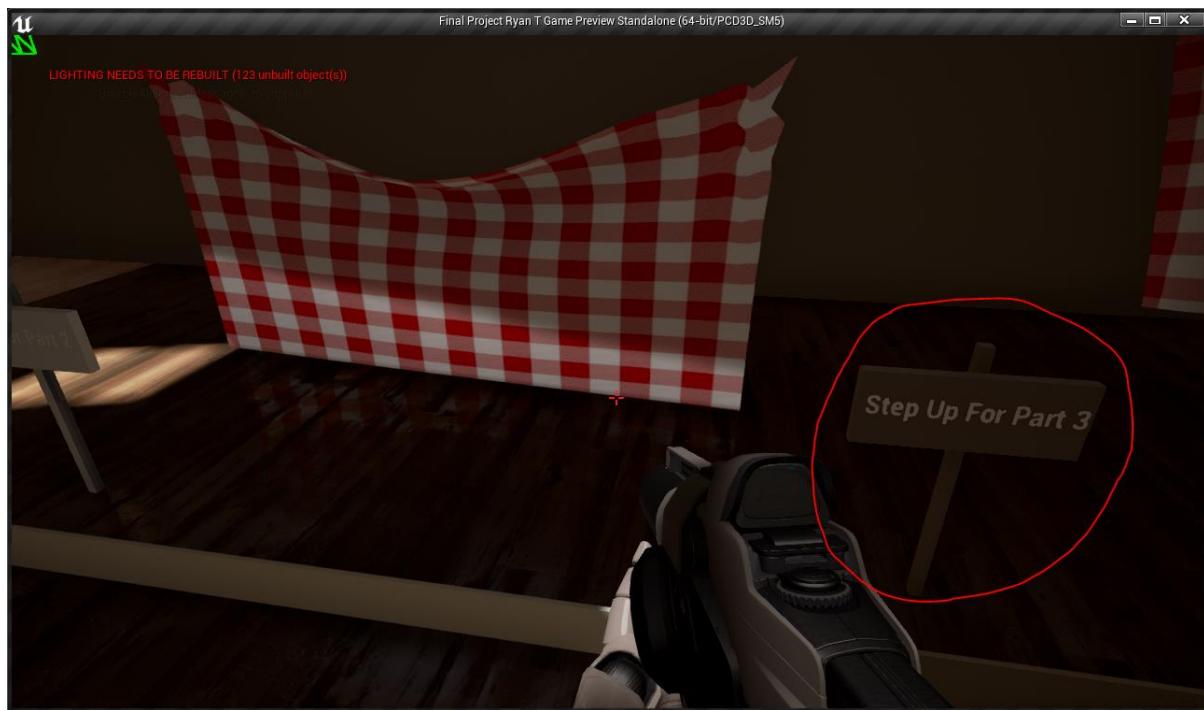


Figure 430

Added sign posts to all demos that are user trigger able.



Fig

Placeholder fountain.



Figure 431

Added final fountain.



Figure 432

Shrunk lightmass importance volume.

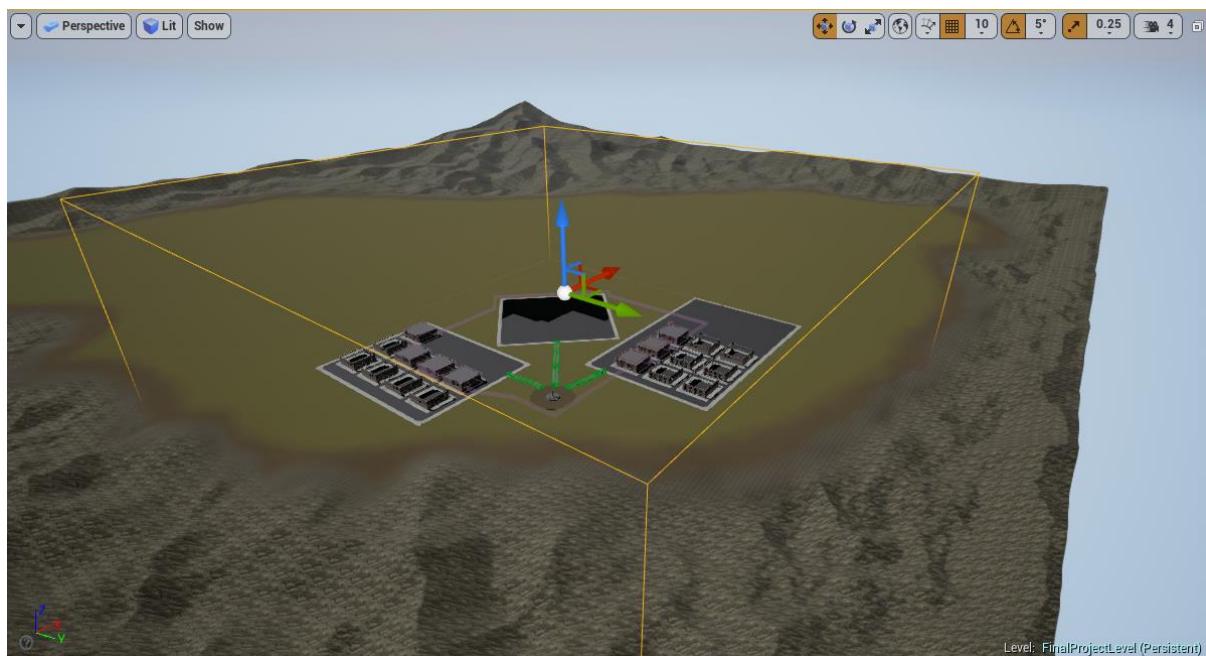


Figure 433

Added sign to area.

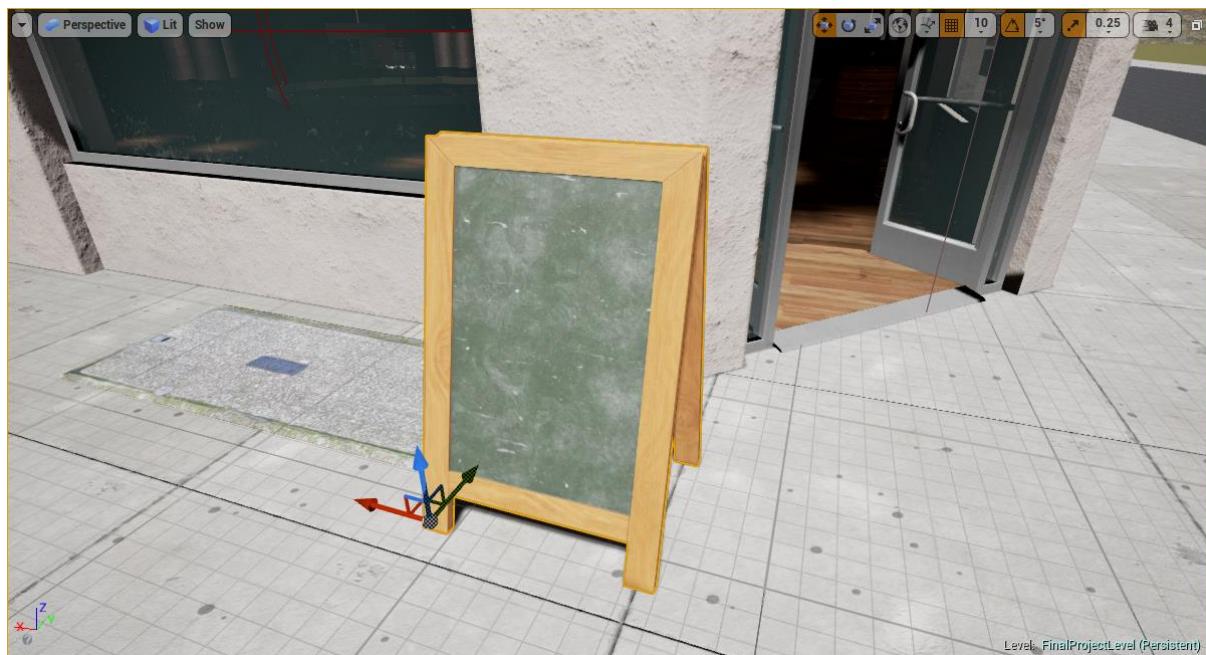


Figure 434

Added text, and moved sign.



Figure 435

Added signs to all areas. Also assed sign to paths with lights

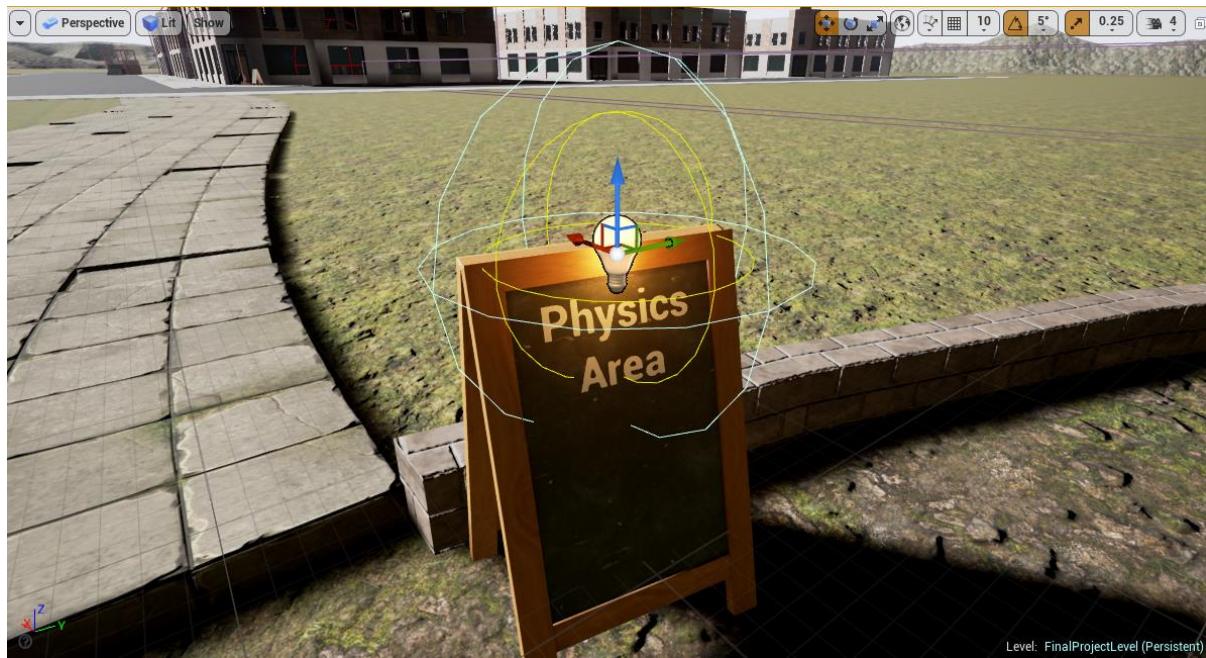


Figure 436

## 6.6. Projectiles

Creating new StickProjectile Actor.

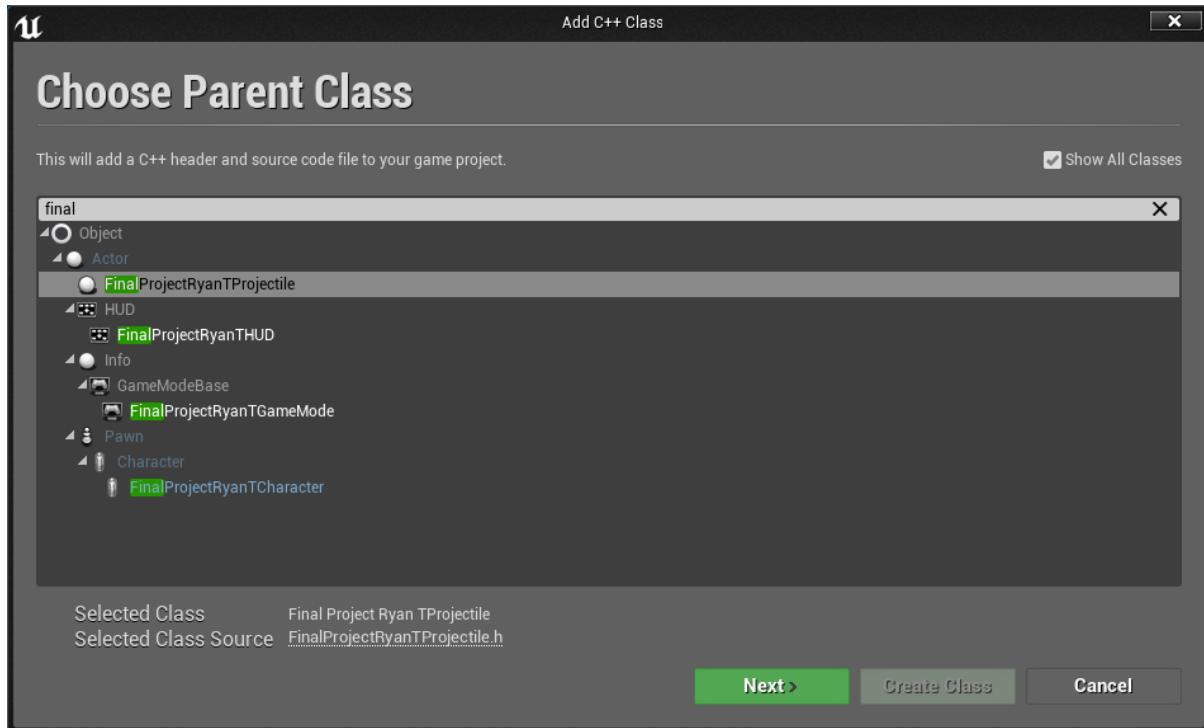


Figure 437

Creating new StickProjectile Actor.

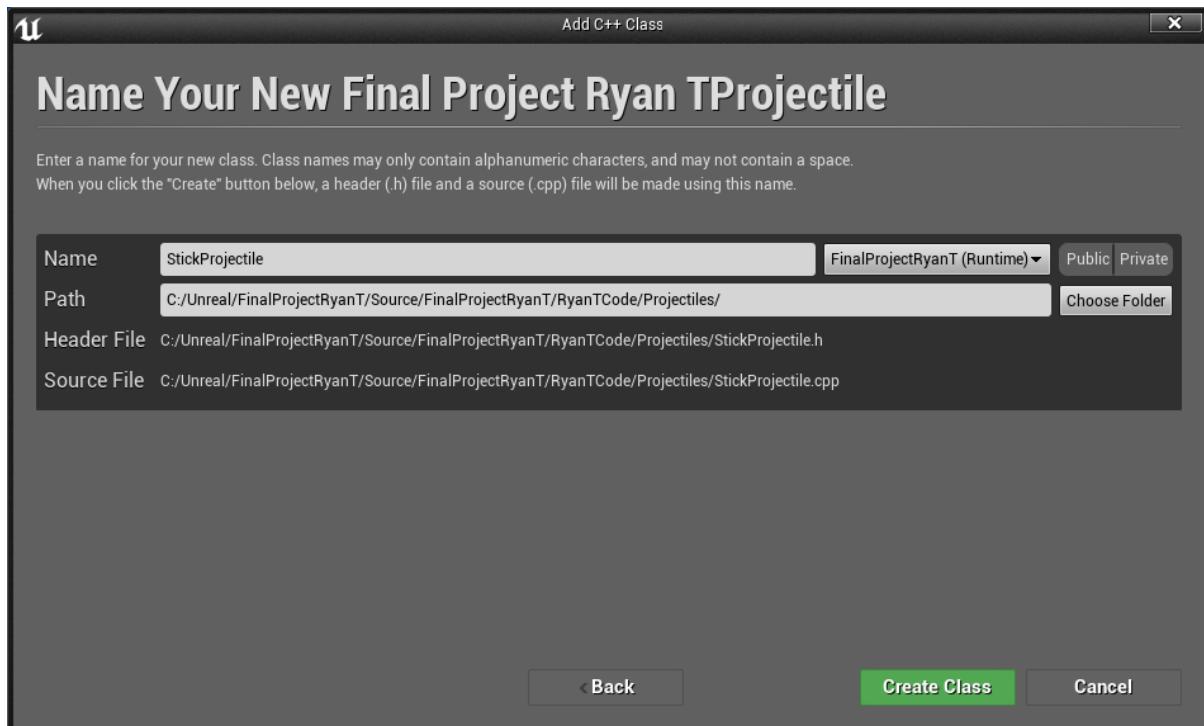


Figure 438

### Creating new StickProjectile Actor (CPP file)

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.  
2  
3 #include "FinalProjectRyanT.h"  
4 #include "StickProjectile.h"  
5  
6  
7  
8  
9
```

Figure 439

### Creating new StickProjectile Actor (Header file).

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.  
2  
3 #pragma once  
4  
5 #include "FinalProjectRyanTProjectile.h"  
6 #include "StickProjectile.generated.h"  
7  
8 /**  
9 *  
10 */  
11 UCCLASS()  
12 class FINALPROJECTRYANT_API AStickProjectile : public AFinalProjectRyanTProjectile  
13 {  
14     GENERATED_BODY()  
15  
16  
17  
18 };  
19  
20
```

Figure 440

### Changed FinalProjectRyanTCharacter class to allow a list of projectile types plus current index (Header file)

```
57     /** Gun muzzle's offset from the characters location */  
58     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Gameplay)  
59     FVector GunOffset;  
60  
61     /** Projectile class to spawn */  
62     UPROPERTY(EditDefaultsOnly, Category=Projectile)  
63     // TSubclassOf<class AFinalProjectRyanTProjectile> ProjectileClass;  
64     TArray<TSubclassOf<class AFinalProjectRyanTProjectile>> ProjectileclassList;  
65  
66     int ProjectileClassIndex = 0;  
67  
68     /** Sound to play each time we fire */
```

Figure 441

Changed FinalProjectRyanTCharacter OnFire function to use new list (CPP file).

```
135 void AFinalProjectRyanTCharacter::OnFire()
136 {
137     // try and fire a projectile
138     if (ProjectileClassList[ProjectileClassIndex] != NULL)
139     {
140         UWorld* const World = GetWorld();
141         if (World != NULL)
142         {
143             if (bUsingMotionControllers)
144             {
145                 const FRotator SpawnRotation = VR_MuzzleLocation->GetComponentRotation();
146                 const FVector SpawnLocation = VR_MuzzleLocation->GetComponentLocation();
147                 World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileClassList[ProjectileClassIndex], SpawnLocation, SpawnRotation);
148             }
149             else
150             {
151                 const FRotator SpawnRotation = GetControlRotation();
152                 // MuzzleOffset is in camera space, so transform it to world space before offsetting from the character location to find the final muzzle position
153                 const FVector SpawnLocation = ((FP_MuzzleLocation != nullptr) ? FP_MuzzleLocation->GetComponentLocation() : GetActorLocation()) + SpawnRotation.RotateVector(GunOffset);
154
155                 //Set Spawn Collision Handling Override
156                 FActorSpawnParams ActorSpawnParams;
157                 ActorSpawnParams.SpawnCollisionHandlingOverride = ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDontSpawnIfColliding;
158
159                 // spawn the projectile at the muzzle
160                 World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileClassList[ProjectileClassIndex], SpawnLocation, SpawnRotation, ActorSpawnParams);
161             }
162         }
163     }
164
165     // try and play the sound if specified
166     if (FireSound != NULL)
167     {
168         UGameplayStatics::PlaySoundAtLocation(this, FireSound, GetActorLocation());
169     }
170
171     // try and play a firing animation if specified
172     if (FireAnimation != NULL)
173     {
174         // Get the animation object for the arms mesh
175         UAnimInstance* AnimInstance = Mesh1P->GetAnimInstance();
176         if (AnimInstance != NULL)
177         {
178             AnimInstance->Montage_Play(FireAnimation, 1.f);
179         }
180     }
181 }
182 }
```

Figure 442

Added new OnHit override function to Sticky Projectile that attaches (CPP file).

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "StickProjectile.h"
5 #include "DrawDebugHelpers.h"
6
7 // ****
8 // OnHit - Callback for when projectile hits another actor or component
9 // ****
10 void ASTickProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
11 {
12     // Only add impulse and destroy projectile if we hit a physics
13     if ((OtherActor != NULL) && (OtherActor != this) && (OtherComp != NULL))
14     {
15         // Attach the projectile to whatever it hits
16         AttachToActor(OtherActor, FAttachmentTransformRules::KeepWorldTransform);
17
18         // Add a physics impulse to the other actor to make it react
19         if (OtherComp->IsSimulatingPhysics())
20             OtherComp->AddImpulseAtLocation(GetVelocity() * 100.0f, GetActorLocation());
21
22         UE_LOG(LogTemp, Warning, TEXT("hit %s "), *OtherComp->GetName());
23         // Destroy();
24     }
25 }
```

Figure 443

Enter and exit callbacks implemented. For now they just change to fixed projectiles on entry.

```
44 //*****
45 // EnterVolume - Called when component enters collision volume
46 //*****
47 void AProjectileVolume::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
48 {
49     // Only allowed if overlapped by any kind of character
50     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
51         return;
52
53     // Get a pointer to the player character
54     AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);
55
56     if ( !Player )
57         return;
58
59     Player->ProjectileClassIndex      = eProjectileType::Attract;
60     Player->ProjectileClassIndexRight = eProjectileType::Repel;
61 }
62
63 //*****
64 // ExitVolume - Called when component exits collision volume
65 //*****
66 void AProjectileVolume::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
67 {
68     // Only allowed if overlapped by any kind of character
69     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
70         return;
71
72     // Get a pointer to the player character
73     AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);
74
75     if ( !Player )
76         return;
77
78     Player->ProjectileClassIndex      = eProjectileType::Default;
79     Player->ProjectileClassIndexRight = eProjectileType::None;
80 }
```

Figure 444

Added new OnHit override function to Sticky Projectile that attaches (H file)

```
8 /**
9  * 
10 */
11 UCLASS()
12 class FINALPROJECTRYANT_API ASTickProjectile : public AFinalProjectRyanTProjectile
13 {
14     GENERATED_BODY()
15
16     /* called when projectile hits something */
17     UFUNCTION()
18     virtual void OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit) override;
19 };
20
```

Figure 445

Added 2 new projectile blueprints (Atract and Repel) for flocking demo.

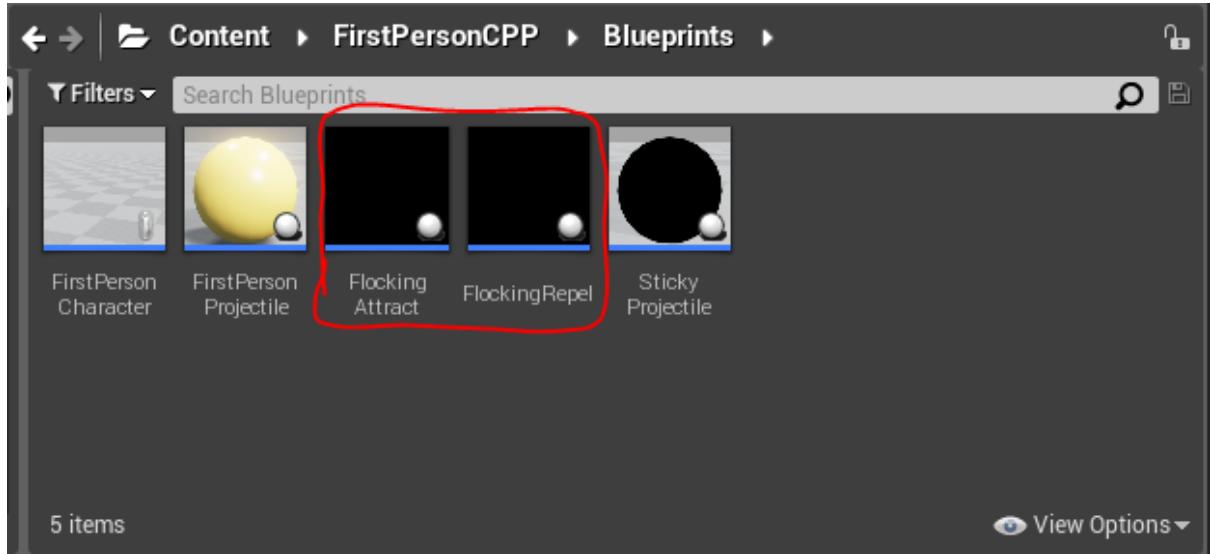


Figure 446

Changed colour of FlockingAttract Projectile to Green.

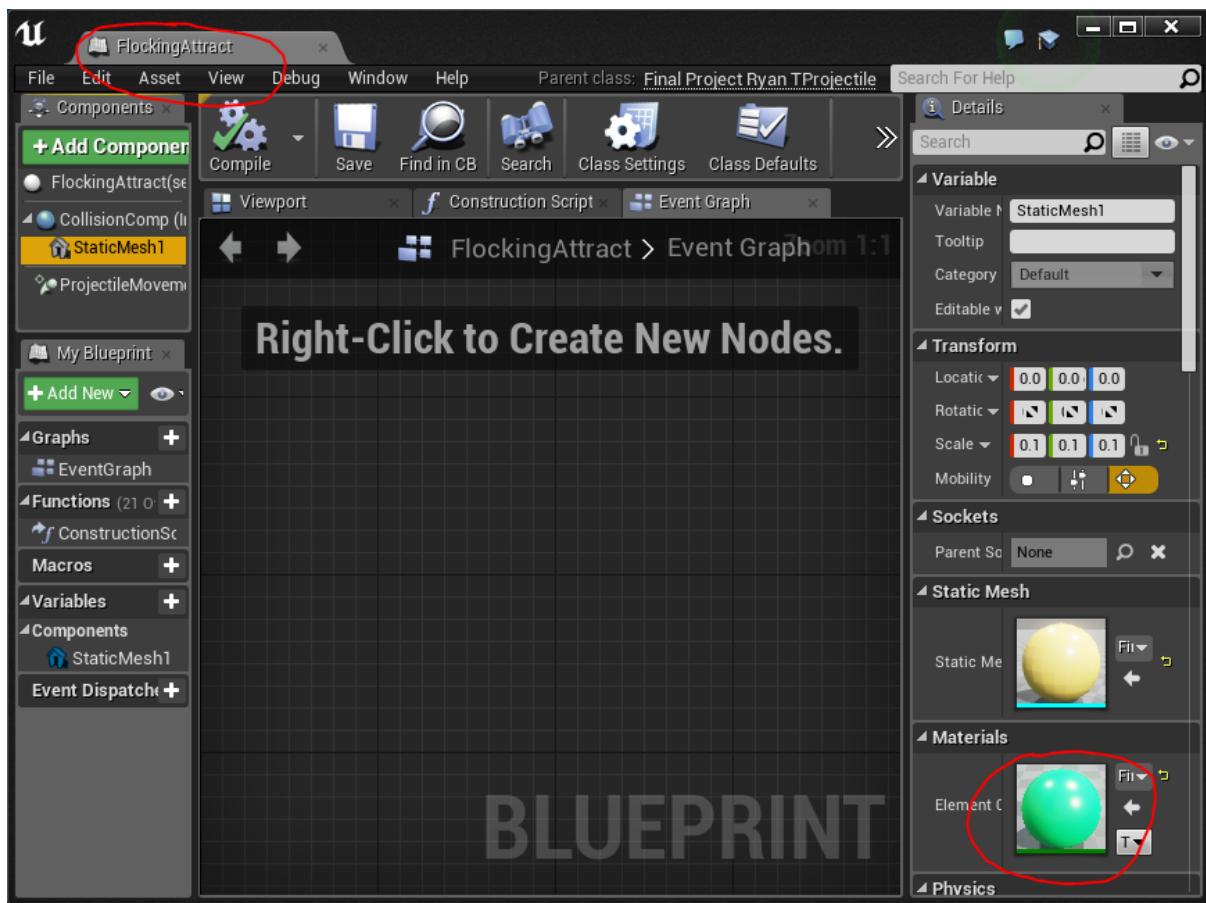


Figure 447

Changed colour of FlockingRepel Projectile to Red.

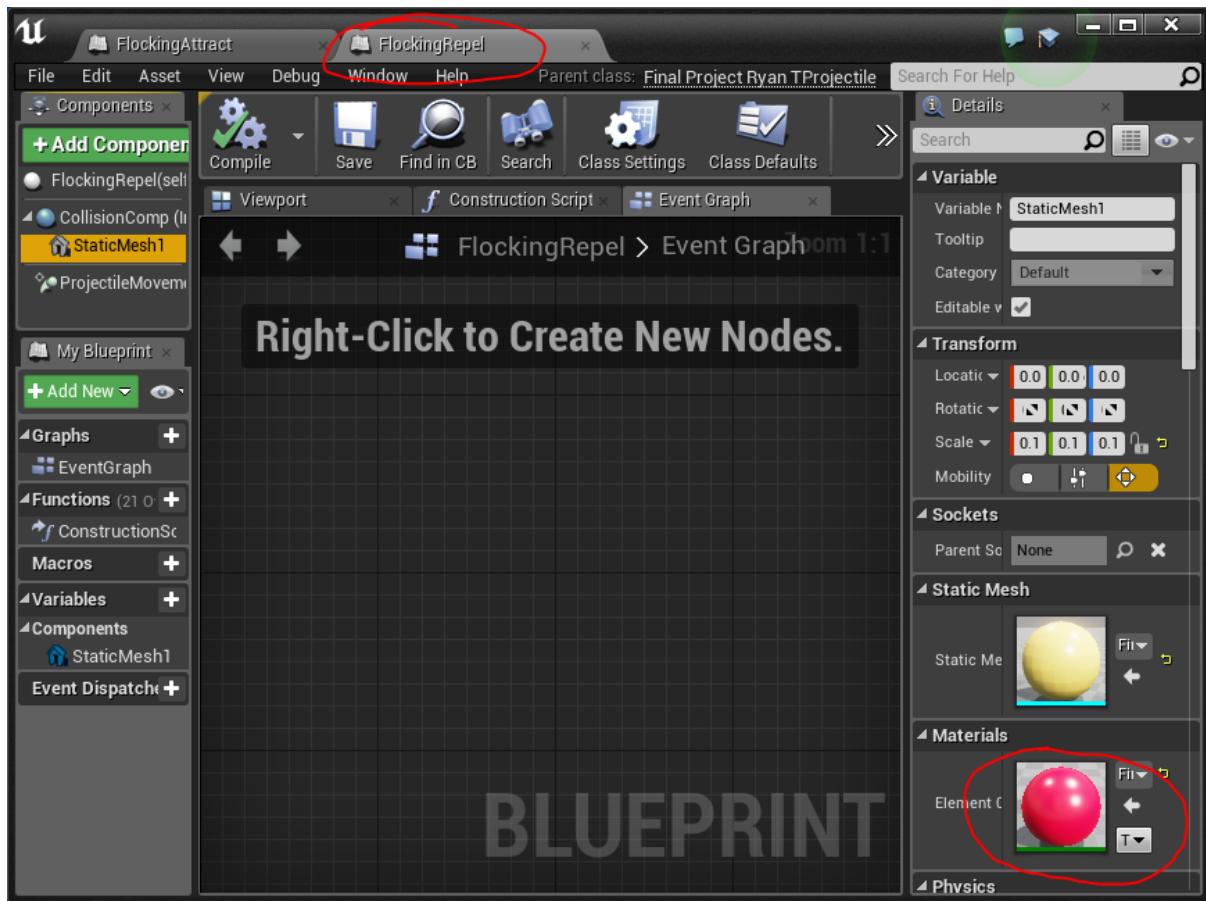


Figure 448

Added them to the projectile list in the characters blueprint.

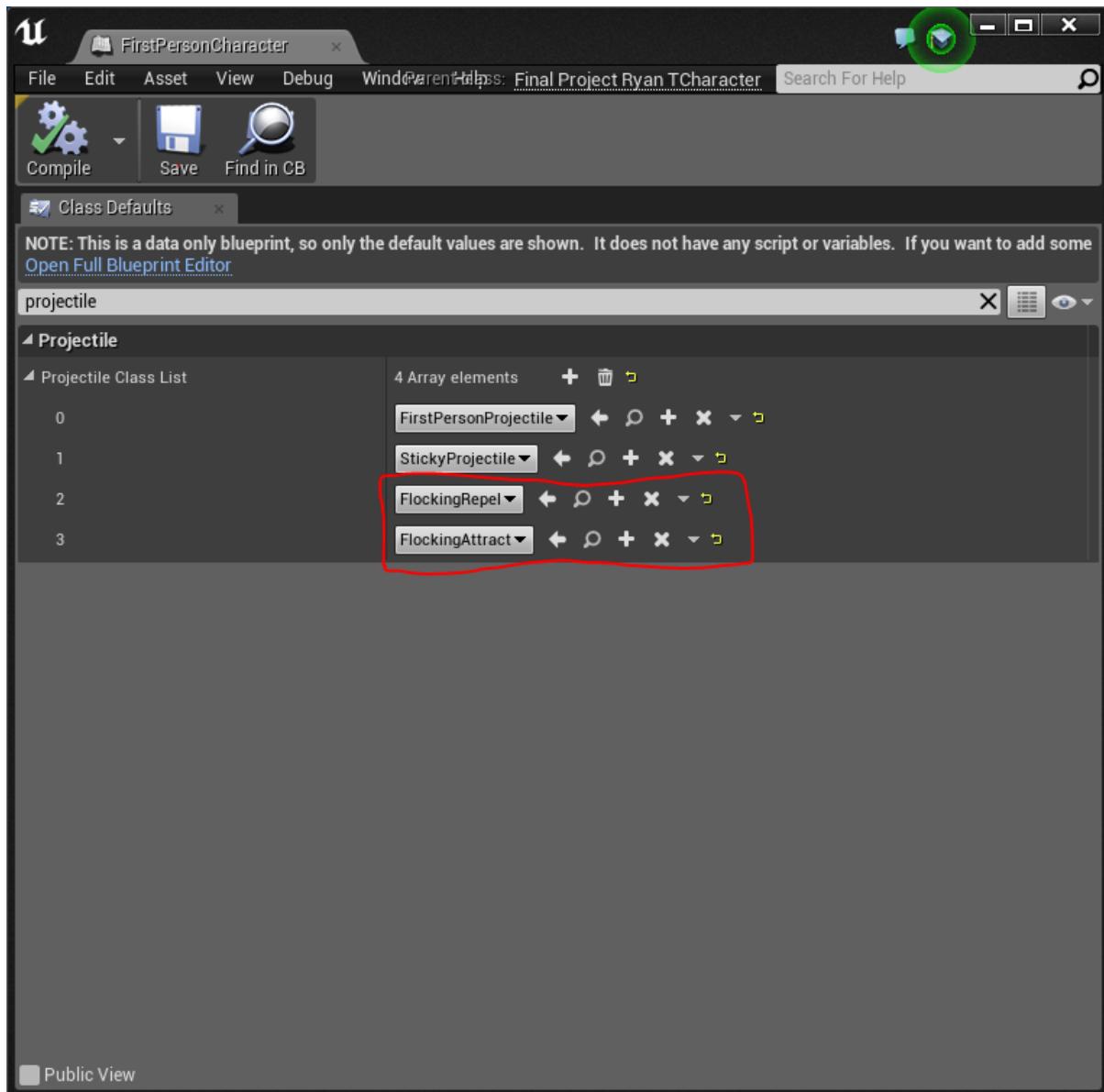


Figure 449

Added code to characters OnFire function. If ProjectileIndex is less than zero then he has no weapon, so don't fire anything.

```
135 void AFinalProjectRyanTCharacter::OnFire()
136 {
137     if ( ProjectileClassIndex < 0 )
138         return; // Player has no weapon, so just return
139
140     // try and fire a projectile
141     if (ProjectileClassList[ProjectileClassIndex] != NULL)
142     {
143         UWorld* const World = GetWorld();
144         if (World != NULL)
145         {
146             if (bUsingMotionControllers)
147             {
148                 const FRotator SpawnRotation = VR_MuzzleLocation->GetComponentRotation();
149                 const FVector SpawnLocation = VR_MuzzleLocation->GetComponentLocation();
150                 World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileClassList[ProjectileClassIndex], SpawnLocation, SpawnRotation);
151             }
152         }
153     }
154 }
```

Figure 450

Added Type eEnum and Type member variable to projectile.

```
5     UENUM() // Tell Unreal Editor that this enum is exposed for editing in its details panels
6     enum class eProjectileType : uint8
7     {
8         Default,
9         Sticky,
10        Repel,
11        Attract,
12    };
13
14
15    UCCLASS(config=Game)
16    class AFinalProjectRyanTProjectile : public AActor
17    {
18        GENERATED_BODY()
19
20        /** Sphere collision component */
21        UPROPERTY(VisibleDefaultsOnly, Category=Projectile)
22        class USphereComponent* CollisionComp;
23
24        /** Projectile movement component */
25        UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
26        class UProjectileMovementComponent* ProjectileMovement;
27
28        // Called when the game starts or when spawned
29        virtual void BeginPlay() override;
30
31        // Called when the game ends or when deleted
32        virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;
33
34    public:
35        AFinalProjectRyanTProjectile();
36
37        /** called when projectile hits something */
38        UFUNCTION()
39        virtual void OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit);
40
41        /** Returns CollisionComp subobject */
42        FORCEINLINE class USphereComponent* GetCollisionComp() const { return CollisionComp; }
43        /** Returns ProjectileMovement subobject */
44        FORCEINLINE class UProjectileMovementComponent* GetProjectileMovement() const { return ProjectileMovement; }
45
46        static TArray<class AFinalProjectRyanTProjectile*> ProjectileList;
47
48        eProjectileType Type = eProjectileType::Default;
49    };
50
```

Figure 451

Added Code to character OnFire to set the fired projectiles type.

```
170    // try and fire a projectile
171    if (ProjectileClassList[ProjectileClassIndex] != NULL)
172    {
173        UWorld* const World = GetWorld();
174        if (World != NULL)
175        {
176            AFinalProjectRyanTProjectile* Projectile = nullptr;
177
178            if (!bUsingMotionControllers)
179            {
180                const FRotator SpawnRotation = VR_MuzzleLocation->GetComponentRotation();
181                const FVector SpawnLocation = VR_MuzzleLocation->GetComponentLocation();
182                Projectile = World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileClassList[ProjectileClassIndex], SpawnLocation, SpawnRotation);
183            }
184            else
185            {
186                const FRotator SpawnRotation = GetControlRotation();
187                // MuzzleOffset is in camera space, so transform it to world space before offsetting from the character location to find the final muzzle position
188                const FVector SpawnLocation = (FP_MuzzleLocation != nullptr) ? FP_MuzzleLocation->GetComponentLocation() : GetActorLocation() + SpawnRotation.RotateVector(GunOffset);
189
190                //Set Spawn Collision Handling Override
191                FActorSpawnParameters ActorSpawnParams;
192                ActorSpawnParams.SpawnCollisionHandlingOverride = ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDontSpawnIfColliding;
193
194                // spawn the projectile at the muzzle
195                Projectile = World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileClassList[ProjectileClassIndex], SpawnLocation, SpawnRotation, ActorSpawnParams);
196            }
197
198            // Set the type in the projectile
199            if ( Projectile )
200                Projectile->Type = eProjectileType(ProjectileClassIndex);
201        }
202    }
203
```

Figure 452

Added new input action in project settings (FireRight) for firing a second projectile.

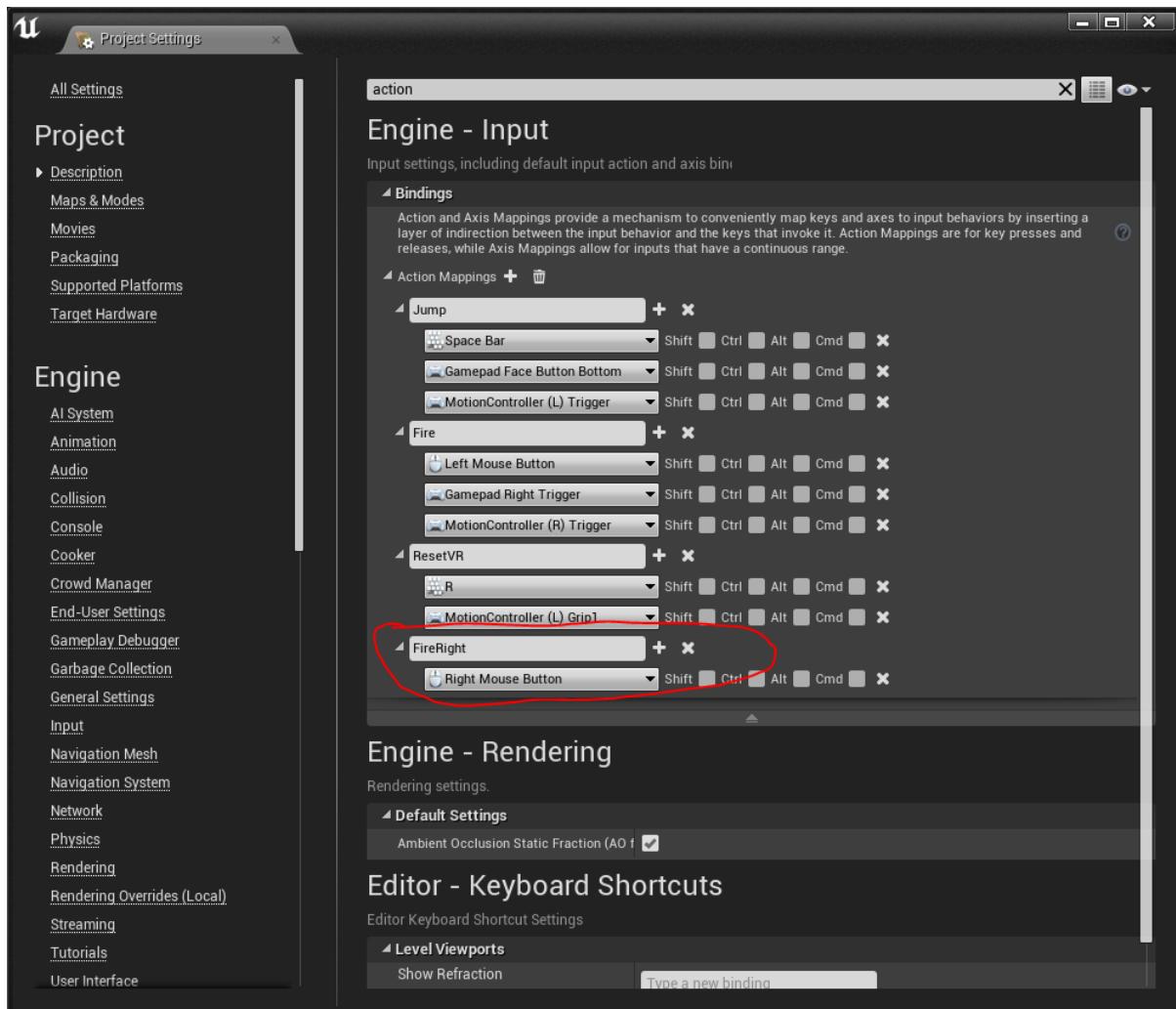


Figure 453

Added 2nd projectile index to character class for right click firing.

```

65
66     int ProjectileClassIndex = 0;           // Default Weapon
67     int ProjectileClassIndexRight = -1;      // Not equipped by default
68
69     /** Sound to play each time we fire */
70     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Gameplay)
71     class USoundBase* FireSound;
72
73     /** AnimMontage to play each time we fire */
74     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay)
75     class UAnimMontage* FireAnimation;
76
77     /** Whether to use motion controller location for aiming. */
78     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay)
79     uint32 bUsingMotionControllers : 1;
80

```

Figure 454

Added an `OnFireRight` callback function (Copy of `OnFire` but using the new right index member).

```
192
193     void AFinalProjectRyanTCharacter::onFireRight()
194     {
195         if ( ProjectileClassIndexRight < 0 )
196             return; // Player has no weapon, so just return
197
198         // try and fire a projectile
199         if (ProjectileclassList[ProjectileClassIndexRight] == NULL)
200         {
201             UWorld* const World = GetWorld();
202             if (World != NULL)
203             {
204                 AFinalProjectRyanTProjectile* Projectile = nullptr;
205
206                 if (busingMotionControllers)
207                 {
208                     const FRotator SpawnRotation = VR_MuzzleLocation->GetComponentRotation();
209                     const FVector SpawnLocation = VR_MuzzleLocation->GetComponentLocation();
210                     Projectile = World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileclassList[ProjectileClassIndexRight], SpawnLocation, SpawnRotation);
211                 }
212                 else
213                 {
214                     const FRotator SpawnRotation = GetControlRotation();
215                     // MuzzleOffset is in camera space, so transform it to world space before offsetting from the character location to find the final muzzle position
216                     const FVector SpawnLocation = ((FP_MuzzleLocation != nullptr) ? FP_MuzzleLocation->GetComponentLocation() : GetActorLocation()) + SpawnRotation.RotateVector(GunOffset);
217
218                     //Set Spawn Collision Handling Override
219                     FActorSpawnParameters ActorSpawnParams;
220                     ActorSpawnParams.SpawnCollisionHandlingOverride = ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDontSpawnIfColliding;
221
222                     // spawn the projectile at the muzzle
223                     Projectile = World->SpawnActor<AFinalProjectRyanTProjectile>(ProjectileclassList[ProjectileClassIndexRight], SpawnLocation, SpawnRotation, ActorSpawnParams);
224                 }
225
226                 // Set the type in the projectile
227                 if ( Projectile )
228                     Projectile->Type = AFinalProjectRyanTProjectile::eType(ProjectileClassIndexRight);
229             }
230
231             // try and play the sound if specified
232             if (FireSound != NULL)
233             {
234                 UGameplayStatics::PlaySoundAtLocation(this, FireSound, GetActorLocation());
235             }
236
237             // try and play a firing animation if specified
238             if (FireAnimation != NULL)
239             {
240                 // Get the animation object for the arms mesh
241                 UAnimInstance* AnimInstance = Mesh1P->GetAnimInstance();
242                 if (AnimInstance != NULL)
243                 {
244                     AnimInstance->Montage_Play(FireAnimation, 1.f);
245                 }
246             }
247         }
248     }
```

*Figure 455*

Added binding for player input to bine 'FireRight' to the OnFireRightFunction.

```
107 void AFinalProjectRyanTCharacter::SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
108 {
109     // set up gameplay key bindings
110     check(PlayerInputComponent);
111
112     PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &ACharacter::Jump);
113     PlayerInputComponent->BindAction("Jump", IE_Released, this, &ACharacter::StopJumping);
114
115     //InputComponent->BindTouch(EInputEvent::IE_Pressed, this, &AFinalProjectRyanTCharacter::TouchStarted);
116     if (EnableTouchscreenMovement(PlayerInputComponent) == false)
117     {
118         PlayerInputComponent->BindAction("Fire", IE_Pressed, this, &AFinalProjectRyanTCharacter::OnFire);
119         PlayerInputComponent->BindAction("FireRight", IE_Pressed, this, &AFinalProjectRyanTCharacter::OnFireRight);
120     }
121
122     PlayerInputComponent->BindAction("ResetVR", IE_Pressed, this, &AFinalProjectRyanTCharacter::OnResetVR);
123
124     PlayerInputComponent->BindAxis("MoveForward", this, &AFinalProjectRyanTCharacter::MoveForward);
125     PlayerInputComponent->BindAxis("MoveRight", this, &AFinalProjectRyanTCharacter::MoveRight);
126
127     // We have 2 versions of the rotation bindings to handle different kinds of devices differently
128     // "turn" handles devices that provide an absolute delta, such as a mouse.
129     // "turnrate" is for devices that we choose to treat as a rate of change, such as an analog joystick
130     PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);
131     PlayerInputComponent->BindAxis("TurnRate", this, &AFinalProjectRyanTCharacter::TurnAtRate);
132     PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);
133     PlayerInputComponent->BindAxis("LookUpRate", this, &AFinalProjectRyanTCharacter::LookUpAtRate);
134 }
```

*Figure 456*

Character can now fire 2 weapons. One on left mouse button and one on Right mouse button.

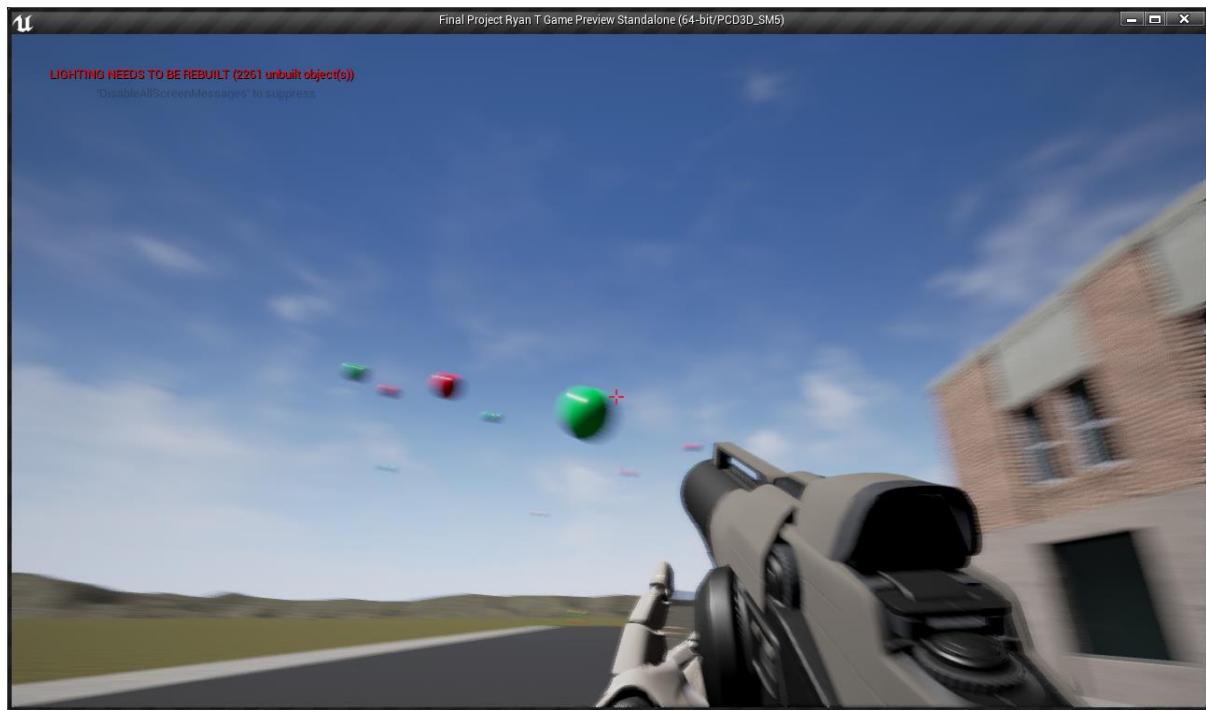


Figure 457

Added function to display equipped projectiles to HUD code.

```
96 // ****
97 // DrawProjectileText - Displays description text to show projectiles that are equipped
98 // ****
99 bool AFinalProjectRyanTHUD::DrawProjectileText()
100 {
101     // Get a pointer to the player character
102     AFinalProjectRyanTCharacter* Player = Cast( GetWorld()->GetFirstPlayerController()->GetPawn() );
103     if ( !Player )
104         return false;
105
106     FString String = "Projectiles:";  

107
108     if ( Player->ProjectileClassIndex >= 0 )
109     {
110         String.Append( "\n" );
111         String.Append( "    LButton: " );
112         String.Append( Player->GetProjectileName(0) );
113     }
114
115     if ( Player->ProjectileClassIndexRight >= 0 )
116     {
117         String.Append( "\n" );
118         String.Append( "    RButton: " );
119         String.Append( Player->GetProjectileName(1) );
120     }
121
122     // Position of text
123     FVector2D Pos(Canvas->ClipX - 240, 10 );
124
125     // Draw the text
126     FCanvasTextItem TextItem( Pos, FText::FromString(String), Font, FLinearColor(0.8f, 0.8f, 0.8f, 1.0f) );
127
128     // Enable drop shadow
129     TextItem.FontRenderInfo.bEnableShadow = true;
130     TextItem.ShadowColor      = FLinearColor(0.0f, 0.0f, 0.0f, 1.0f);
131     TextItem.ShadowOffset    = FVector2D( 2.0f, 2.0f );
132
133     TextItem.BlendMode       = SE_BLEND_Translucent;
134     TextItem.Scale           = FVector2D( 1.0f, 1.0f );
135
136     Canvas->DrawItem( TextItem );
137     return true;
138 }
```

Figure 458

Called said function from DrawHUD.

```
31 // ****
32 // DrawHUD - Draw whole HUD
33 // ****
34 void AFinalProjectRyanTHUD::DrawHUD()
35 {
36     Super::DrawHUD();
37
38     // Draw very simple crosshair
39
40     // find center of the Canvas
41     const FVector2D Center(Canvas->ClipX * 0.5f, Canvas->ClipY * 0.5f);
42
43     // offset by half the texture's dimensions so that the center of the texture aligns with the center of the Canvas
44     const FVector2D CrosshairDrawPosition( (Center.X),
45                                         (Center.Y + 20.0f));
46
47     // Draw the sub-title box
48     if ( !DrawSubTitleText() )
49     {
50         // Draw the crosshair
51         FCanvasTileItem TileItem( CrosshairDrawPosition, CrosshairTex->Resource, FLinearColor::White );
52         TileItem.BlendMode = SE_BLEND_Translucent;
53         Canvas->DrawItem( TileItem );
54
55         // Draw text for equipped Projectiles
56         DrawProjectileText();
57     }
58 }
```

Figure 459

Added function to character class to get name of equipped projectiles.

```
137 // ****
138 // GetProjectileName - Get the name of the equipped projectile for left or right button
139 // ****
140 FString AFinalProjectRyanTCharacter::GetProjectileName( int LeftOrRight )
141 {
142     // 1 = Right 0 = Left
143     int Index = LeftOrRight ? ProjectileClassIndexRight : ProjectileClassIndex;
144
145     if ( Index < 0 )
146         return FString( "None" );
147
148     switch ( Index )
149     {
150         case eProjectileType::Default:
151             return FString("Default");
152
153         case eProjectileType::Sticky:
154             return FString("Sticky");
155
156         case eProjectileType::Attract:
157             return FString("Attract");
158
159         case eProjectileType::Repel:
160             return FString("Repel");
161     }
162     return FString("Error");
163 }
164 }
```

Figure 460

Creating a New Projectile volume actor. This will be used to change the equiped projectiles.

The image consists of two screenshots of the Unreal Engine's 'Add C++ Class' dialog box.

**Screenshot 1: Choose Parent Class**

This screen shows a list of parent classes:

- Character**: A character is a type of Pawn that includes the ability to walk around.
- Pawn**: A Pawn is an actor that can be 'possessed' and receive input from a controller.
- Actor**: An Actor is an object that can be placed or spawned in the world. This option is highlighted with a yellow background.
- Actor Component**: An ActorComponent is a reusable component that can be added to any actor.
- Scene Component**: A Scene Component is a component that has a scene transform and can be attached to other scene components.

Below the list are buttons for 'Selected Class' (set to 'Actor') and 'Selected Class Source' (set to 'Actor.h'). At the bottom are 'Next >', 'Create Class', and 'Cancel' buttons.

**Screenshot 2: Name Your New Actor**

This screen asks for the name of the new class:

Name: `ProjectileVolume`

Path: `C:/Unreal/FinalProjectRyanT/Source/FinalProjectRyanT/RyanTCode/Projectiles`

Header File: `C:/Unreal/FinalProjectRyanT/Source/FinalProjectRyanT/RyanTCode/Projectiles/ProjectileVolume_h`

Source File: `C:/Unreal/FinalProjectRyanT/Source/FinalProjectRyanT/RyanTCode/Projectiles/ProjectileVolume_cpp`

At the bottom are 'Back', 'Create Class', and 'Cancel' buttons.

Figure 461

Creating a New Projectile volume actor. This will be used to change the equipped projectiles.

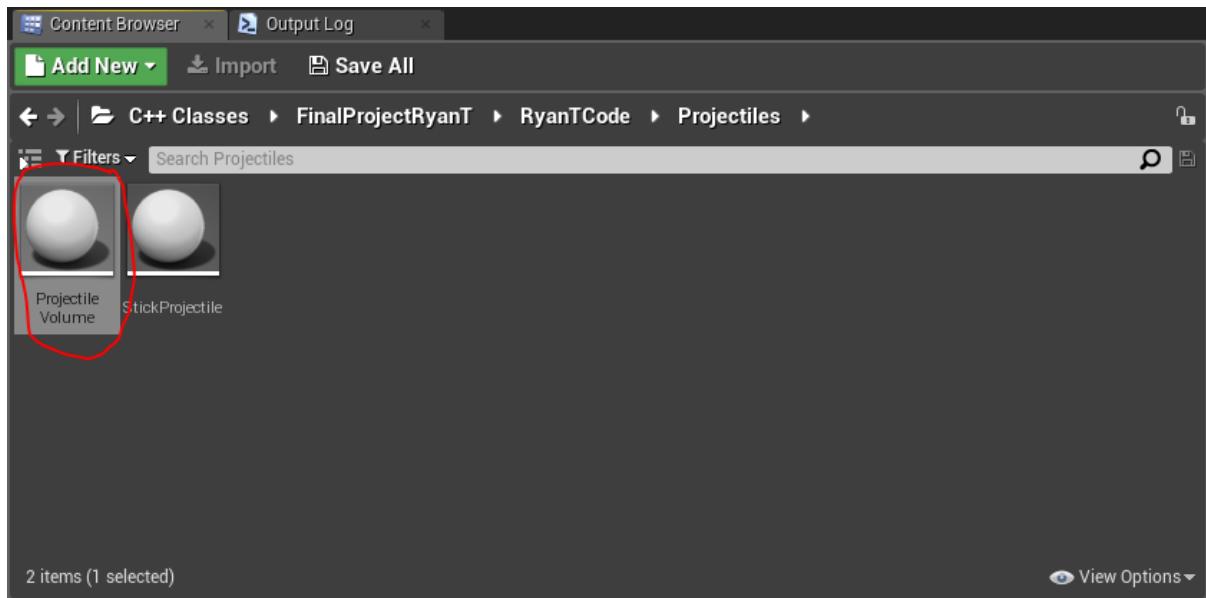


Figure 462

Initial code.

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "ProjectileVolume.h"
5
6
7 // Sets default values
8 AProjectileVolume::AProjectileVolume()
9 {
10     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
11     PrimaryActorTick.bCanEverTick = true;
12 }
13
14 // Called when the game starts or when spawned
15 void AProjectileVolume::BeginPlay()
16 {
17     Super::BeginPlay();
18 }
19
20 // Called every frame
21 void AProjectileVolume::Tick( float DeltaTime )
22 {
23     Super::Tick( DeltaTime );
24 }
25
26
27
28 |
```

```

1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "ProjectileVolume.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API AProjectileVolume : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     AProjectileVolume();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23
24
25 };
26

```

Figure 463

An instance of new volume in scene.

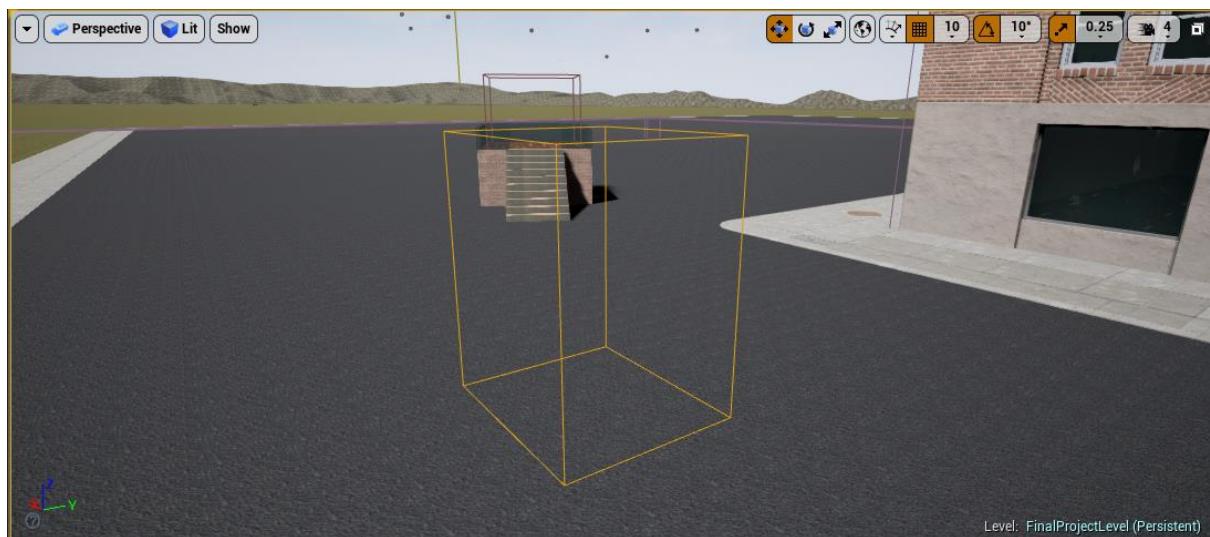


Figure 464

Added a box component pointer an enter and exit volume callback functions to H.

```
8  UCLASS()
9  class FINALPROJECTRYANT_API AProjectileVolume : public AActor
10 {
11     GENERATED_BODY()
12
13     public:
14         // Sets default values for this actor's properties
15         AProjectileVolume();
16
17         // Called when the game starts or when spawned
18         virtual void BeginPlay() override;
19
20         // Called every frame
21         virtual void Tick( float DeltaSeconds ) override;
22
23         UPROPERTY(EditAnywhere)
24         UBoxComponent* TriggerVolume = nullptr;
25
26         UFUNCTION()
27         void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
28         UFUNCTION()
29         void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
30     };

```

Figure 465

Created Box Component in constructor.

```
// ****
// Constructor
// ****
AProjectileVolume::AProjectileVolume()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));

    RootComponent = TriggerVolume;
}
```

Figure 466

Register callbacks with box component.

```
22 // ****
23 // BeginPlay - Called when the game starts or when spawned
24 // ****
25 void AProjectileVolume::BeginPlay()
26 {
    Super::BeginPlay();

    TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);

    TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AProjectileVolume::EnterVolume);
    TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AProjectileVolume::ExitVolume);
}

```

Figure 467

When out of volume.

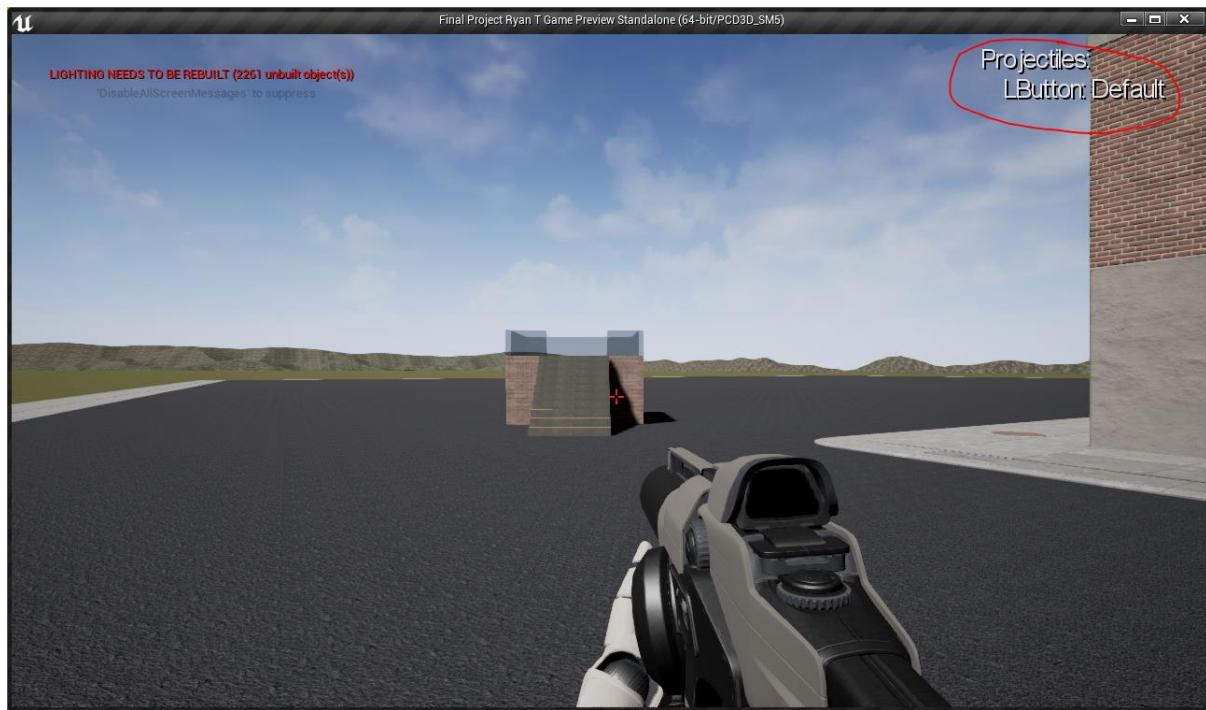


Figure 468

When Inside volume.



Figure 469

Added left and right projectile type UPROPERTY to ProjectileVolume. Now each volume can set its types.

```

8  UClass()
9  class FINALPROJECTRYANT_API AProjectileVolume : public AActor
10 {
11     GENERATED_BODY()
12
13     public:
14         // Sets default values for this actor's properties
15         AProjectileVolume();
16
17         // Called when the game starts or when spawned
18         virtual void BeginPlay() override;
19
20         // Called every frame
21         virtual void Tick( float DeltaSeconds ) override;
22
23         UPROPERTY(EditAnywhere)
24             UBoxComponent* TriggerVolume = nullptr;
25
26         UPROPERTY(EditAnywhere, Category = "Projectiles")
27             eProjectileType LeftButtonType = eProjectileType::None;
28
29         UPROPERTY(EditAnywhere, Category = "Projectiles")
30             eProjectileType RightButtonType = eProjectileType::None;
31
32         UFUNCTION()
33         void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
34         UFUNCTION()
35         void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
36     };
37

```

Figure 470

Types set in editor.

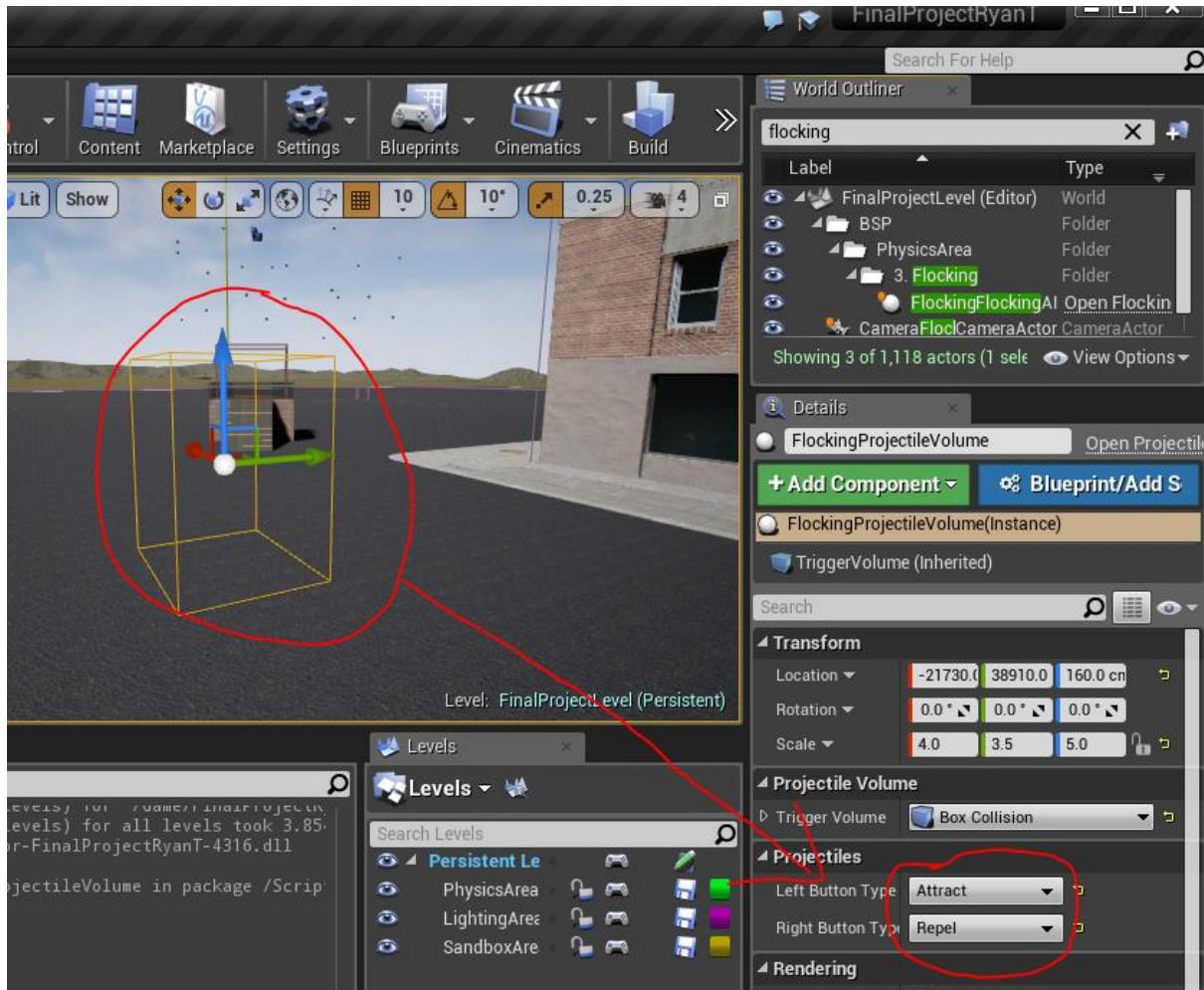


Figure 471

Added member variables to the header file for remembering old projectile types on entry.

```
25     UPROPERTY(EditAnywhere, Category = "Projectiles")
26         eProjectileType LeftButtonType      = eProjectileType::None;
27
28
29     UPROPERTY(EditAnywhere, Category = "Projectiles")
30         eProjectileType RightButtonType    = eProjectileType::None;
31
32     // For remembering setting on entry
33     eProjectileType OldLeft;
34     eProjectileType OldRight;
35
36     UFUNCTION()
37         void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimi
38     UFUNCTION()
39         void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiv
40     ];
41
```

Figure 472

Code to set the players projectiles from the volumes UPROPERTY members.

```
44 // ****
45 // EnterVolume - Called when component enters collision volume
46 // ****
47 void AProjectileVolume::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other,
48 {
49     // Only allowed if overlapped by any kind of character
50     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
51         return;
52
53     // Get a pointer to the player character
54     AFinalProjectRyanTCharacter* Player = Cast(Other);
55
56     if ( !Player )
57         return;
58
59     // Remember Old Setting
60     OldLeft    = Player->ProjectileClassIndex;
61     OldRight   = Player->ProjectileClassIndexRight;
62
63     // Set players projectile types from this volume
64     Player->ProjectileClassIndex      = LeftButtonType;
65     Player->ProjectileClassIndexRight = RightButtonType;
66 }
67
68 // ****
69 // EnterVolume - Called when component exits collision volume
70 // ****
71 void AProjectileVolume::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimi
72 {
73     // Only allowed if overlapped by any kind of character
74     if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
75         return;
76
77     // Get a pointer to the player character
78     AFinalProjectRyanTCharacter* Player = Cast(Other);
79
80     if ( !Player )
81         return;
82
83     // Restore old settings
84     Player->ProjectileClassIndex      = OldLeft;
85     Player->ProjectileClassIndexRight = OldRight;
86 }
```

Figure 473

Added Projectile Volumes around the Flocking, Collision and Cloth area with relevant Projectiles set.



Figure 474

Changed flocking GetProjectileAvoidDirection to account for new projectile types (Attract and Repel)

```
262 // ****
263 // GetProjectileAvoidDirection - Returns direction away from any close projectiles
264 // ****
265 FVector AFlockingAI::GetProjectileAvoidDirection(FBird& CurrentBird)
266 {
267     FVector ProjectileAvoid(0.0f, 0.0f, 0.0f);
268
269     for (int p = 0; p < AFinalProjectRyanTProjectile::ProjectileList.Num(); p++)
270     {
271         FVector ProjectilePos = AFinalProjectRyanTProjectile::ProjectileList[p]->GetActorLocation();
272         if (FVector::Dist(CurrentBird.Position, ProjectilePos) < AvoidProjectileDistance)
273         {
274             // Get vector from projectile to bird
275             FVector NewDir = ProjectilePos - CurrentBird.Position;
276             NewDir.Normalize();
277
278             switch (AFinalProjectRyanTProjectile::ProjectileList[p]->Type )
279             {
280                 case eProjectileType::Repel:
281                     ProjectileAvoid -= NewDir;           // Move away from projectile
282                     break;
283
284                 case eProjectileType::Attract:
285                     ProjectileAvoid += NewDir;          // Move towards projectile
286                     break;
287             }
288         }
289     }
290
291     return ProjectileAvoid;
292 }
```

Figure 475

## 6.7. Types of Light Building (Lighting Area)

### 6.7.1. Point Light Demo

Creating a new LightDemo actor

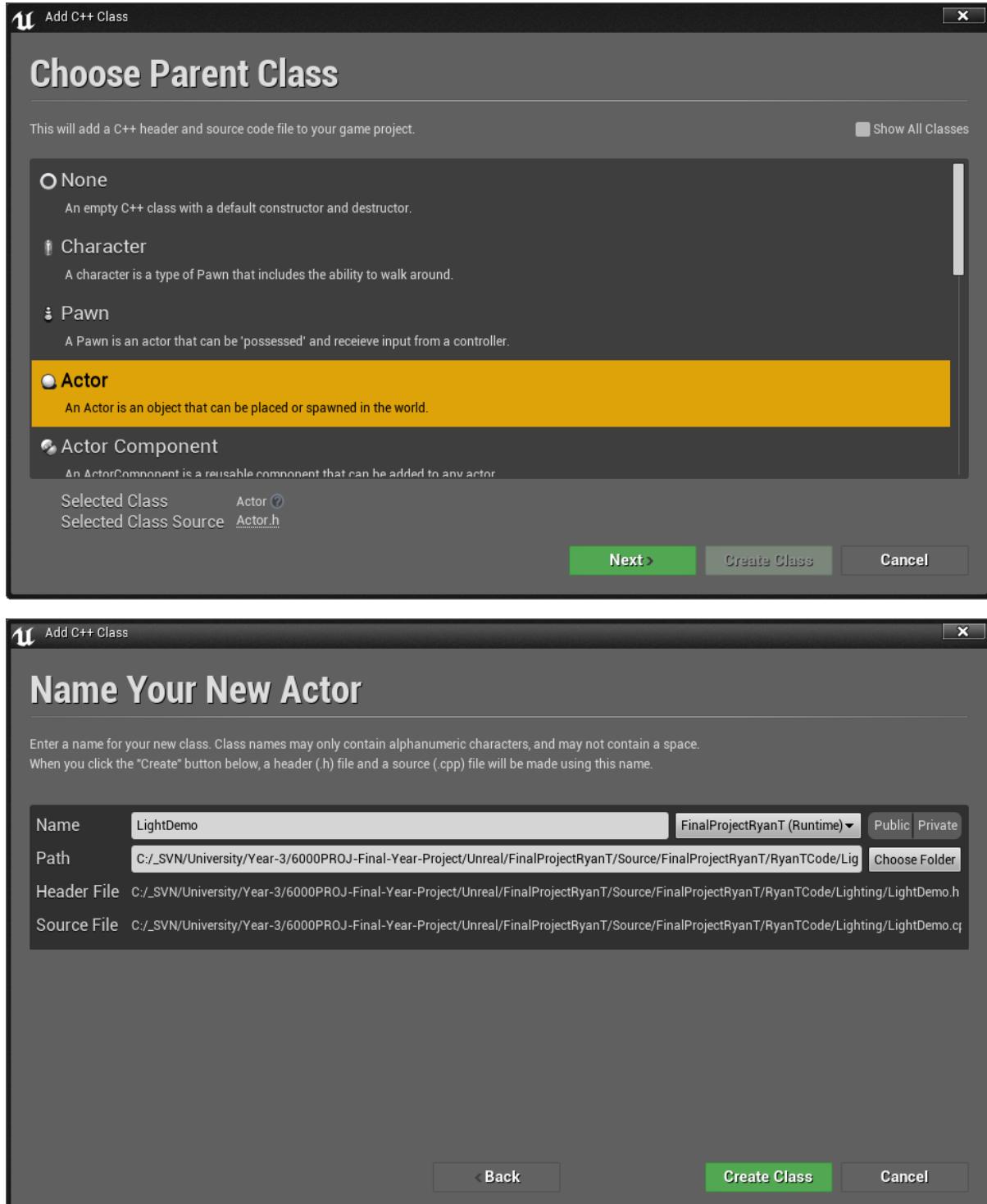


Figure 476

## Initial Code

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #include "FinalProjectRyanT.h"
4 #include "LightDemo.h"
5
6 // Sets default values
7 ALightDemo::ALightDemo()
8 {
9     // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
10    PrimaryActorTick.bCanEverTick = true;
11 }
12
13
14 // Called when the game starts or when spawned
15 void ALightDemo::BeginPlay()
16 {
17     Super::BeginPlay();
18 }
19
20
21 // Called every frame
22 void ALightDemo::Tick( float DeltaTime )
23 {
24     Super::Tick( DeltaTime );
25 }
26
27
```

```
1 // Copyright © 2018 Ryan Tinman, All Rights Reserved.
2
3 #pragma once
4
5 #include "GameFramework/Actor.h"
6 #include "LightDemo.generated.h"
7
8 UCLASS()
9 class FINALPROJECTRYANT_API ALightDemo : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     ALightDemo();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23
24
25 };
26
```

Figure 477

Added Trigger box component pointer and entry, exit callback function to H file.

```
7  UCLASS()
8  class FINALPROJECTRYANT_API ALightDemo : public AActor
9  {
10     GENERATED_BODY()
11
12     public:
13         // Sets default values for this actor's properties
14         ALightDemo();
15
16         // Called when the game starts or when spawned
17         virtual void BeginPlay() override;
18
19         // Called every frame
20         virtual void Tick( float DeltaSeconds ) override;
21
22         UPROPERTY(EditAnywhere)
23         UBoxComponent* TriggerVolume = nullptr;
24
25
26         UFUNCTION()
27         void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& Sweep);
28
29         UFUNCTION()
30         void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
31     };
32
```

Figure 478

Added entry, exit callback function to CPP file.

```
44     // *****
45     // EnterVolume - Called when component enters collision volume
46     // *****
47     void ALightDemo::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& Sweep)
48     {
49         // Only allowed if overlapped by any kind of character
50         if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
51             return;
52
53         // Get a pointer to the player character
54         AFinalProjectRyanTCharacter* Player = Cast(Other);
55
56         if (!Player)
57             return;
58     }
59
60     // *****
61     // ExitVolume - Called when component exits collision volume
62     // *****
63     void ALightDemo::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
64     {
65         // Only allowed if overlapped by any kind of character
66         if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
67             return;
68
69         // Get a pointer to the player character
70         AFinalProjectRyanTCharacter* Player = Cast(Other);
71
72         if (!Player)
73             return;
74     }
```

Figure 479

Created box component in constructor, registerer exit and entry callbacks with box component in BeginPlay.

```
8     // *****
9     // Constructor
10    // *****
11    ALightDemo::ALightDemo()
12    {
13        // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
14        PrimaryActorTick.bCanEverTick = true;
15
16        // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
17        TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));
18
19        RootComponent = TriggerVolume;
20    }
21
22    // *****
23    // BeginPlay - Called when the game starts or when spawned
24    // *****
25    void ALightDemo::BeginPlay()
26    {
27        Super::BeginPlay();
28
29        TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
30        TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
31
32        TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &ALightDemo::EnterVolume);
33        TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &ALightDemo::ExitVolume);
34    }
35
```

Figure 480

Created point light, camera and New LightDemo actor in scene.



Figure 481

Added TextContainer and level sequence functionality to LightDemo Header file.

```

26
27     UPROPERTY()
28     UTextContainerComponent* TextContainer = nullptr;
29
30     UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|Text")
31     float TextContainerIndex = -1; // -1 default = No Text
32
33     UPROPERTY(EditAnywhere, Category = "LightDemo|Trigger")
34     ULevelSequence* LevelSequence = nullptr;
35
36     UPROPERTY()
37     ULevelSequencePlayer* SequencePlayer = nullptr;
38
39     UFUNCTION()
40     void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
41
42     UFUNCTION()
43     void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
44
45
46 private:
47     void OnEndSequenceEvent();
48

```

Figure 482

Added text container Setup code to begin play.

```

22     // ****
23     // BeginPlay - Called when the game starts or when spawned
24     // ****
25     void ALightDemo::BeginPlay()
26     {
27         Super::BeginPlay();
28
29         TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
30         TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
31
32         TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &ALightDemo::EnterVolume);
33         TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &ALightDemo::ExitVolume);
34
35         // Get a list of all UTextContainerComponents
36         TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
37         if (TextComponents.Num() > 0)
38             TextContainer = Cast<UTextContainerComponent>(TextComponents[0]); // Use the first text container component in list
39
40         TextContainerIndex = -1;
41     }

```

Figure 483

Added UpdateSubTitles function and called it from Tick.

```
119 // ****
120 // UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
121 // ****
122 void ALightDemo::UpdateSubTitleText()
123 {
124     if (TextContainer && TextContainerIndex >= 0)
125     {
126         TextContainer->SetTextFromList(-1);           // Turn off 3d text
127
128         FString Temp = TextContainer->GetTextFromList(TextContainerIndex);
129
130         // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
131         while (true)
132         {
133             int32 Index;
134             if (!Temp.FindChar(TCHAR('\r'), Index))
135                 break;
136             Temp.RemoveAt(Index, 1);
137         }
138         AFinalProjectRyanTHUD::SetString(Temp);
139     }
140 }
```

Figure 484

Added code to trigger volume exit and entry functions to start and stop level sequence and reset HUD subtitle text.

```
57 // ****
58 // EnterVolume - Called when component enters collision volume
59 // ****
60 void ALightDemo::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
61 {
62     // Only allowed if overlapped by any kind of character
63     if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
64         return;
65
66     // Get a pointer to the player character
67     AFinalProjectRyanTCharacter* Player = Cast(Other);
68
69     if (!Player)
70         return;
71
72     if (LevelSequence)
73     {
74         FLevelSequencePlaybackSettings Settings;
75
76         Settings.LoopCount = 0;
77
78         SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
79         SequencePlayer->Play();
80     }
81 }
82
83 // ****
84 // ExitVolume - Called when component exits collision volume
85 // ****
86 void ALightDemo::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
87 {
88     // Only allowed if overlapped by any kind of character
89     if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
90         return;
91
92     // Get a pointer to the player character
93     AFinalProjectRyanTCharacter* Player = Cast(Other);
94
95     if (!Player)
96         return;
97
98     if (SequencePlayer)
99     {
100        SequencePlayer->Stop();
101        SequencePlayer = nullptr;
102    }
103    TextContainerIndex = -1;
104    AFinalProjectRyanTHUD::SetString();
105 }
```

Figure 485

Added TextContainer component to light demo actor and added some initial text.

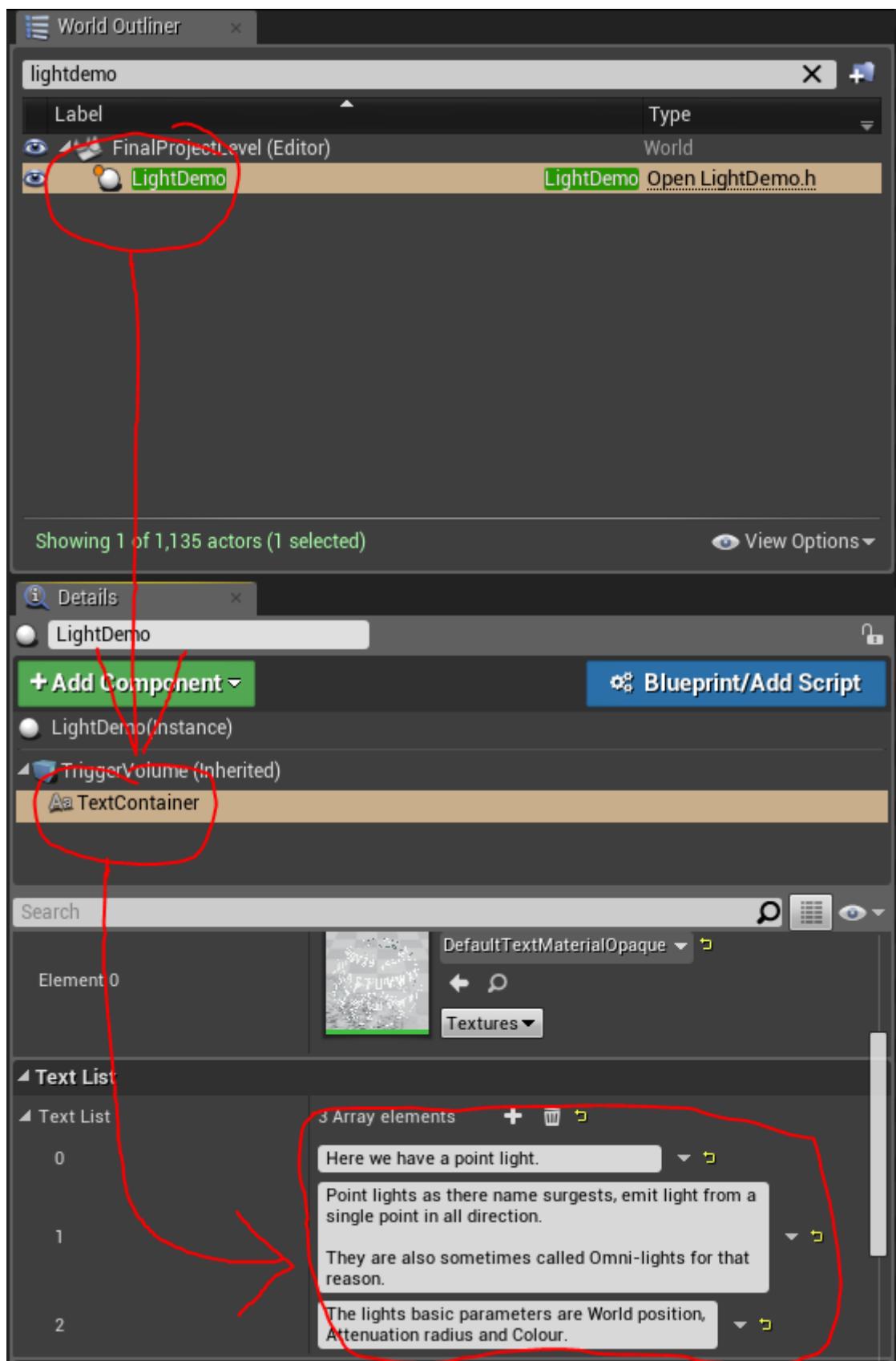


Figure 486

Create new empty level sequence asset to script the point light demo.

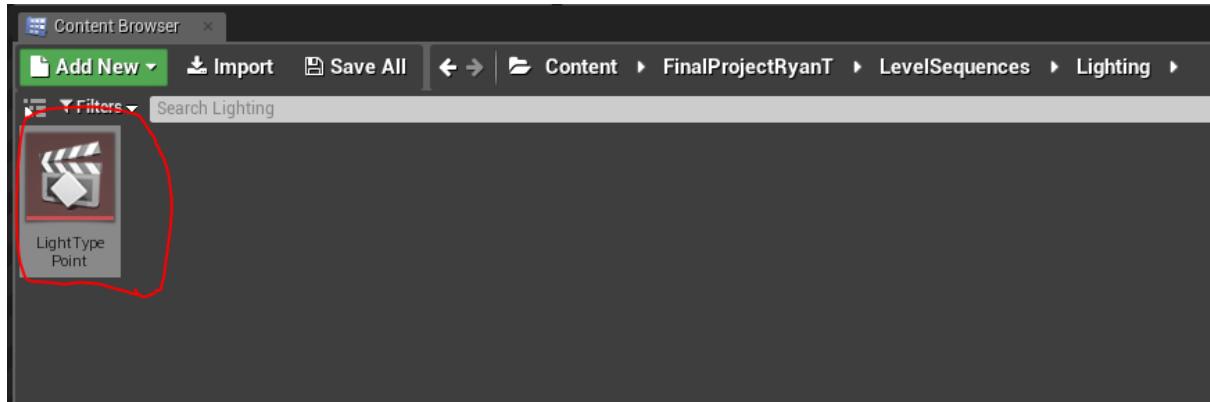


Figure 487

Added some initial relevant Tracks and keys to said level sequence.

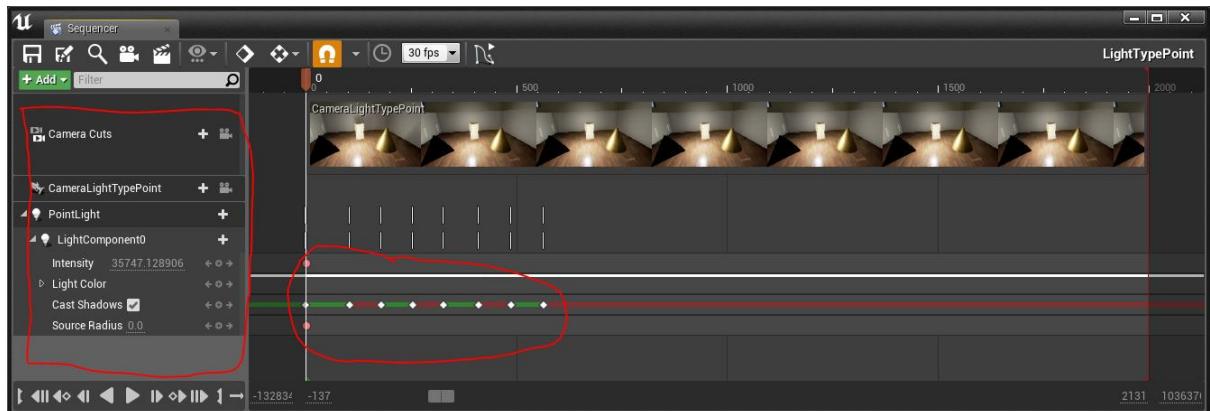


Figure 488

Added Light and Camera actor pointer UPROPERTY to LightDemo actor so that light and camera can be selected from scene in editor.

```
35
36     UPROPERTY(EditAnywhere, Category = "LightDemo|Trigger")
37     ULevelSequence* LevelSequence = nullptr;
38
39     UPROPERTY(EditAnywhere, Category = "LightDemo|Actors")
40     ALight* Light = nullptr;
41
42     UPROPERTY(EditAnywhere, Category = "LightDemo|Actors")
43     ACameraActor* Camera = nullptr;
44
45     UPROPERTY()
46     ULevelSequencePlayer* SequencePlayer = nullptr;
47
48     UFUNCTION()
49     void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPr
50     UFUNCTION()
51     void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiv
52
```

A screenshot of the LightDemo actor's Blueprint code. Several UPROPERTY declarations are highlighted with a red box. These include 'LevelSequence' (category 'LightDemo|Trigger'), 'Light' (category 'LightDemo|Actors'), and 'Camera' (category 'LightDemo|Actors'). The code also includes UFUNCTION declarations for 'EnterVolume' and 'ExitVolume'.

Figure 489

Added bool UPROPERTY bShowLightRadius to allow light volume to be shown from level sequence timeline. Also DebugDrawLight prototype (H file).

```
42     UPROPERTY(EditAnywhere, Category = "LightDemo|Actors")
43     ACameraActor* Camera = nullptr;
44
45     UPROPERTY()
46     ULevelSequencePlayer* SequencePlayer = nullptr;
47
48     UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|Debug")
49     bool bShowLightRadius = false;
50
51     UFUNCTION()
52     void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* Overlapped)
53     UFUNCTION()
54     void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* Overlapped)
55
56 private:
57     void OnEndSequenceEvent();
58     void UpdateSubTitleText();
59     void DebugDrawLight();
60
61 };
62
```

Figure 490

Function to draw various debug info about light. Currently only draws point light radius (Called from Tick).

```
143 //*****
144 // UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
145 // ****
146 void ALightDemo::DebugDrawLight()
147 {
148     if (!Light)
149         return; // No light added so just return
150
151     ULightComponent* LightComp = Light->GetLightComponent();
152
153     if (LightComp->GetClass() == UPointLightComponent::StaticClass())
154     {
155         if (bShowLightRadius)
156         {
157             UPointLightComponent* Point = Cast<UPointLightComponent>(LightComp); // Cast from a light to a point light
158
159             // If it's a point light draw it's radius
160             DrawDebugSphere(GetWorld(), Point->GetComponentLocation(), Point->AttenuationRadius, 32, FColor::White);
161
162         }
163     }
164 }
165
166
```

Figure 491

Completed level sequence for Point Light.

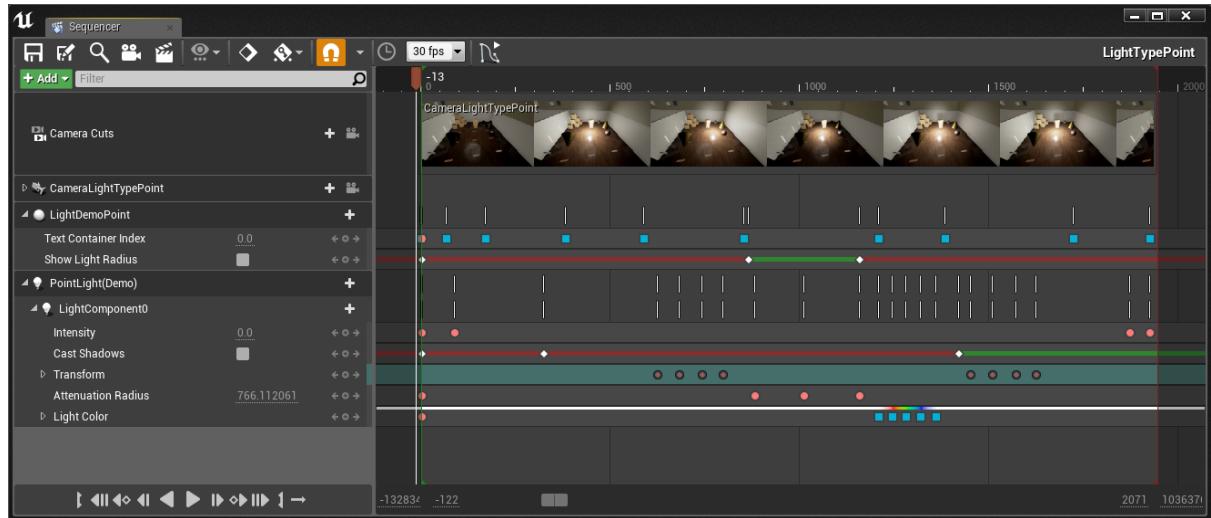


Figure 492

Point Light Sequence: Light is off



Figure 493

## Point Light Sequence: Light Fades up

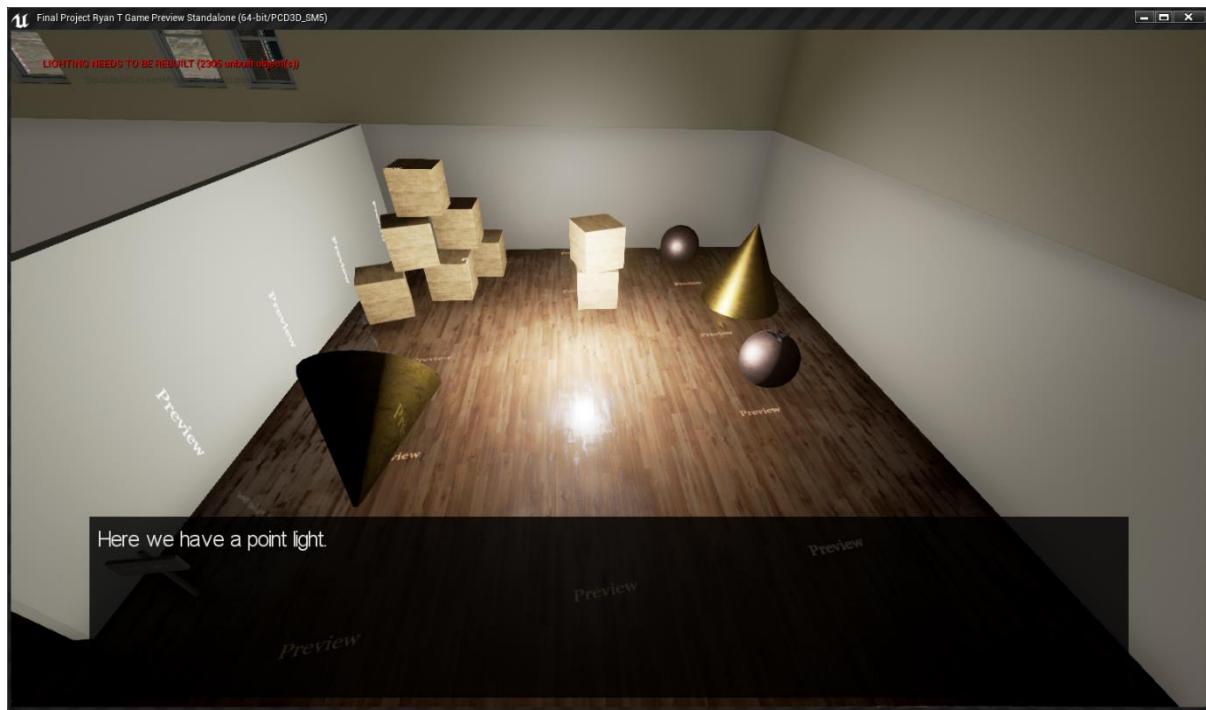


Figure 494

Next text description...



Figure 495

Next text description...

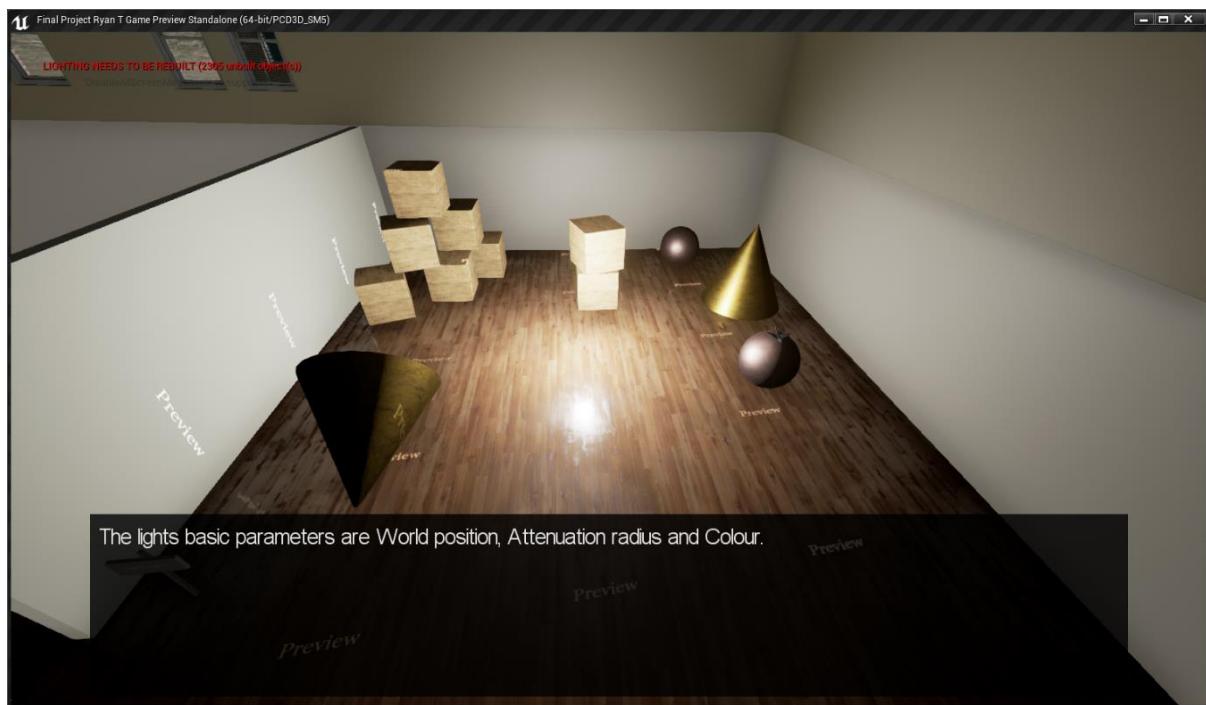


Figure 496

Point Light Sequence: Light move left and right to show light position

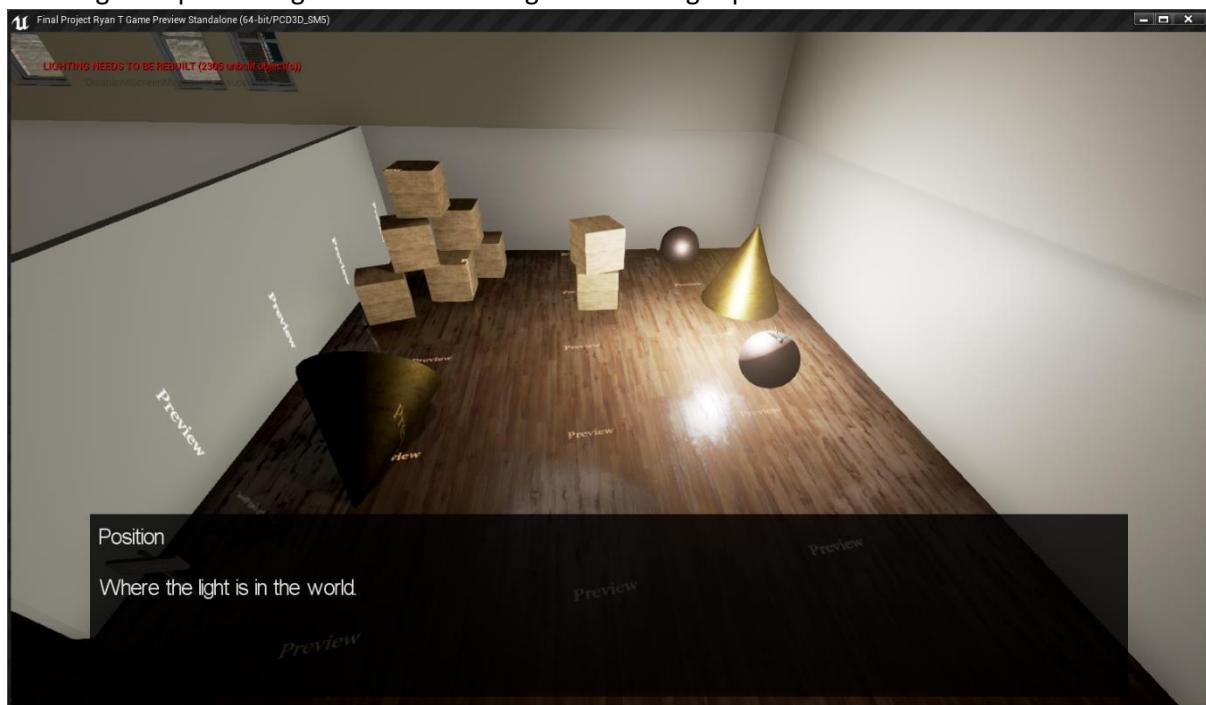


Figure 497

### Point Light Sequence: Light radius grows and shrinks to show radius



Figure 498

### Point Light Sequence: Light colour cycles Red, Green, Blue to show colour

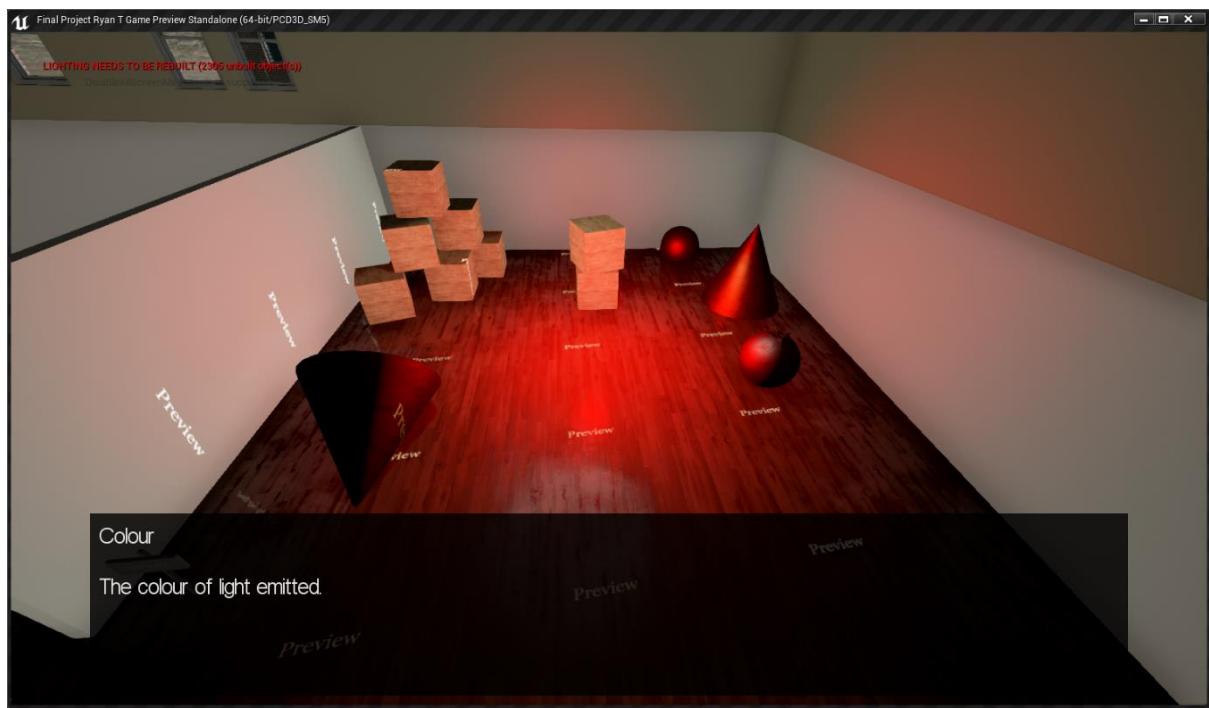


Figure 499

Point Light Sequence: Light moves, shadows can be seen moving



Figure 500

### Point Light Sequence: Light fades back off



Figure 501

#### 6.7.2. Spot Light Demo

Duplicated LightDemoActor and Camera from previous Point Light demo. This will be used as a base for the spot light demo. The text change or replaced.

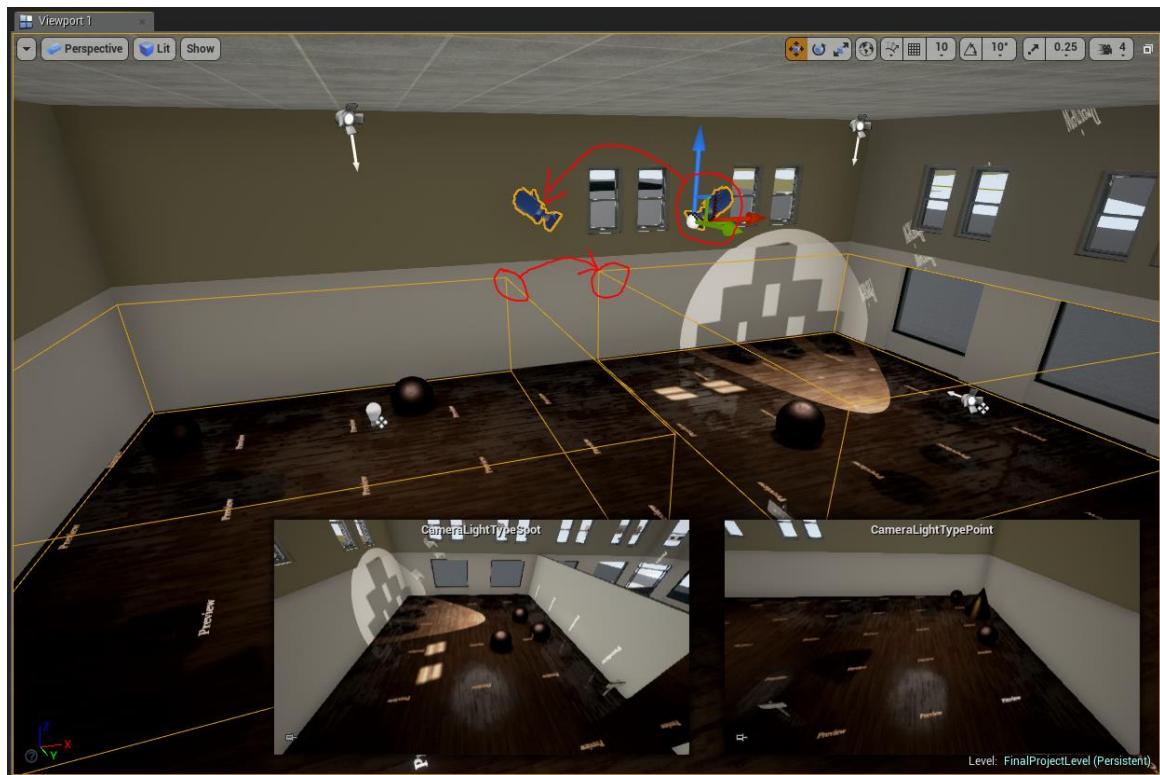


Figure 502

Duplicated the whole of the point light level sequence as a base for the spot light demo.

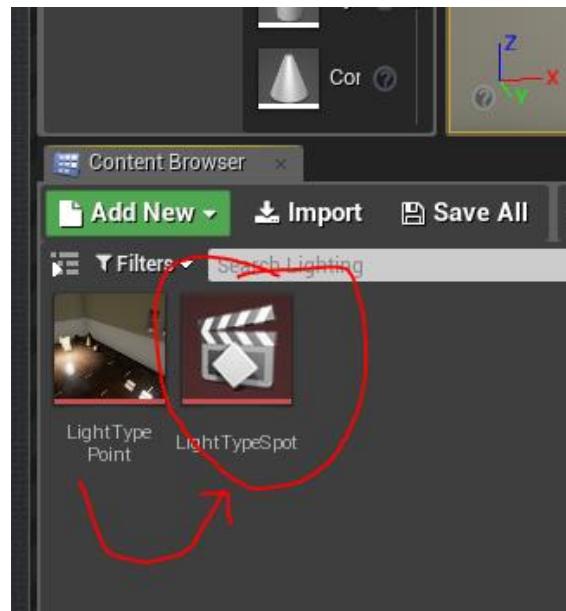


Figure 503

Changed the spot LightDemo actor to point to the newly duplicated level sequence.

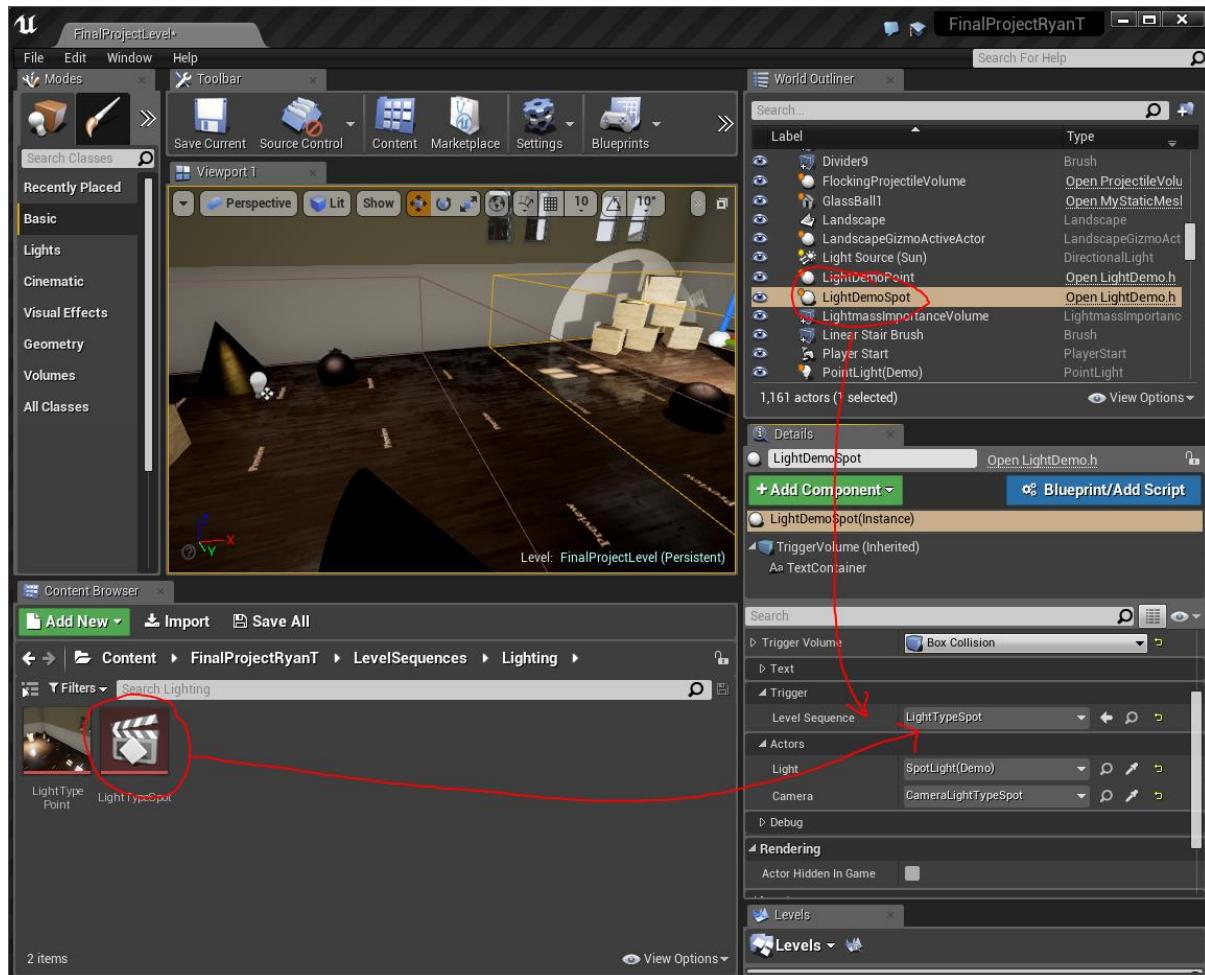
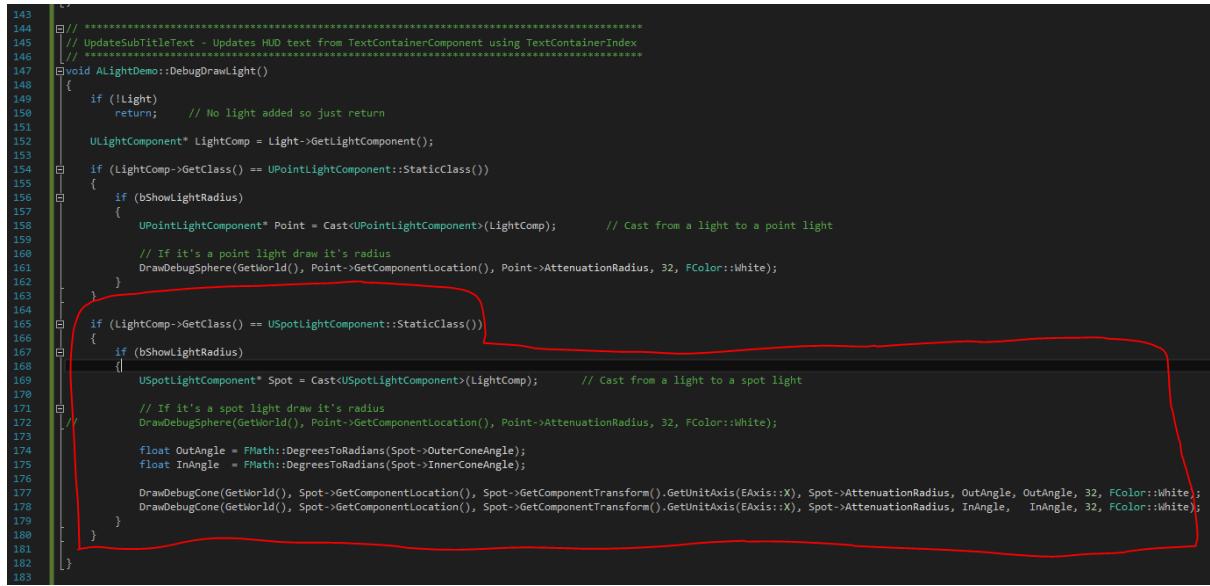


Figure 504

Extended Light Demo actor DebugDrawLight to draw spot light types.



```
143 // ****
144 // UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
145 // ****
146
147 void ALightDemo::DebugDrawLight()
148 {
149     if (!Light)
150         return; // No light added so just return
151
152     ULightComponent* LightComp = Light->GetLightComponent();
153
154     if (LightComp->GetClass() == UPointLightComponent::StaticClass())
155     {
156         if (bShowLightRadius)
157         {
158             UPointLightComponent* Point = Cast<UPointLightComponent>(LightComp); // Cast from a light to a point light
159
160             // If it's a point light draw it's radius
161             DrawDebugSphere(GetWorld(), Point->GetComponentLocation(), Point->AttenuationRadius, 32, FColor::White);
162         }
163     }
164
165     if (LightComp->GetClass() == USpotlightComponent::StaticClass())
166     {
167         if (bShowLightRadius)
168         {
169             USpotlightComponent* Spot = Cast<USpotlightComponent>(LightComp); // Cast from a light to a spot light
170
171             // If it's a spot light draw it's radius
172             DrawDebugSphere(GetWorld(), Point->GetComponentLocation(), Point->AttenuationRadius, 32, FColor::White);
173
174             float OutAngle = FMath::DegreesToRadians(Spot->OuterConeAngle);
175             float InAngle = FMath::DegreesToRadians(Spot->InnerConeAngle);
176
177             DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X), Spot->AttenuationRadius, OutAngle, OutAngle, 32, FColor::White);
178             DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X), Spot->AttenuationRadius, InAngle, InAngle, 32, FColor::White);
179         }
180     }
181 }
182 }
```

Figure 505

Completed Level Sequence for spot lights.

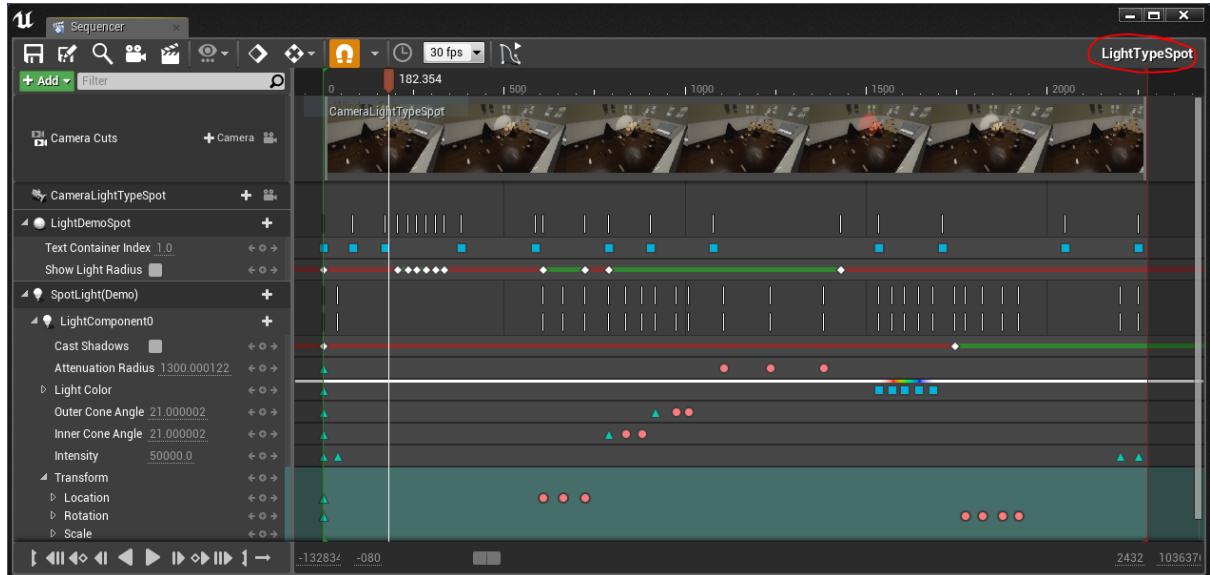


Figure 506

### Spot Light Sequence: Light fades up

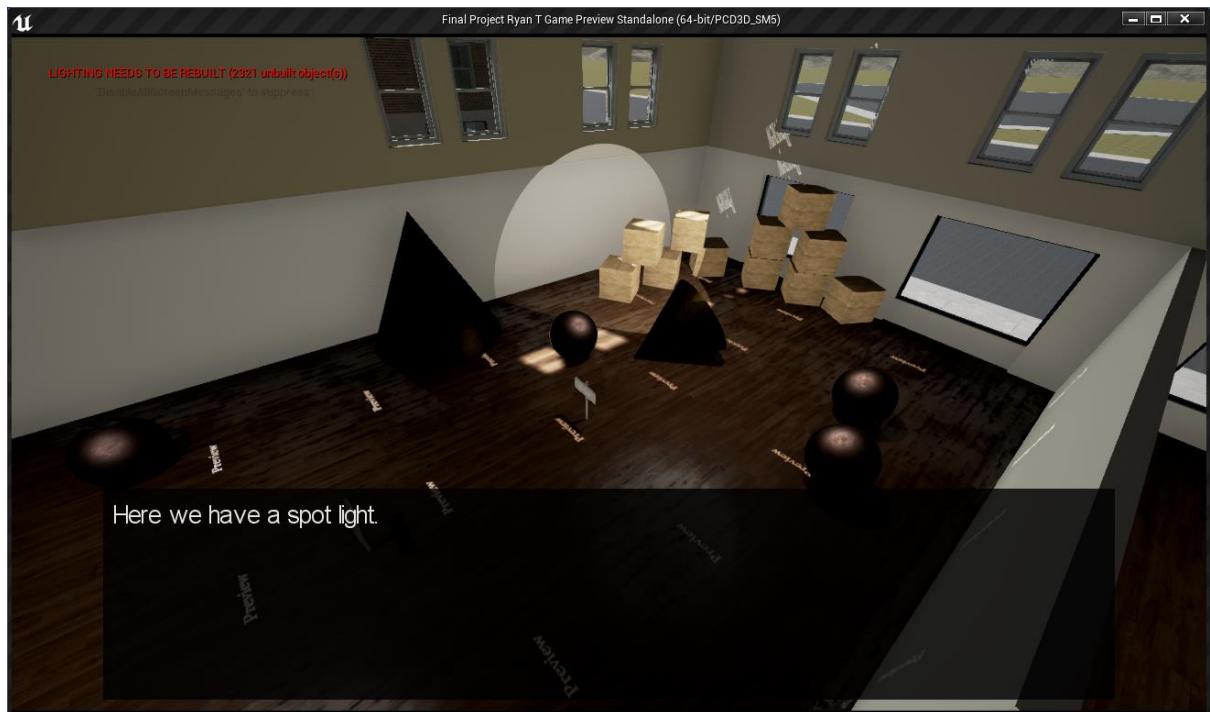


Figure 507

### Spot Light Sequence: Lights volume is highlighted

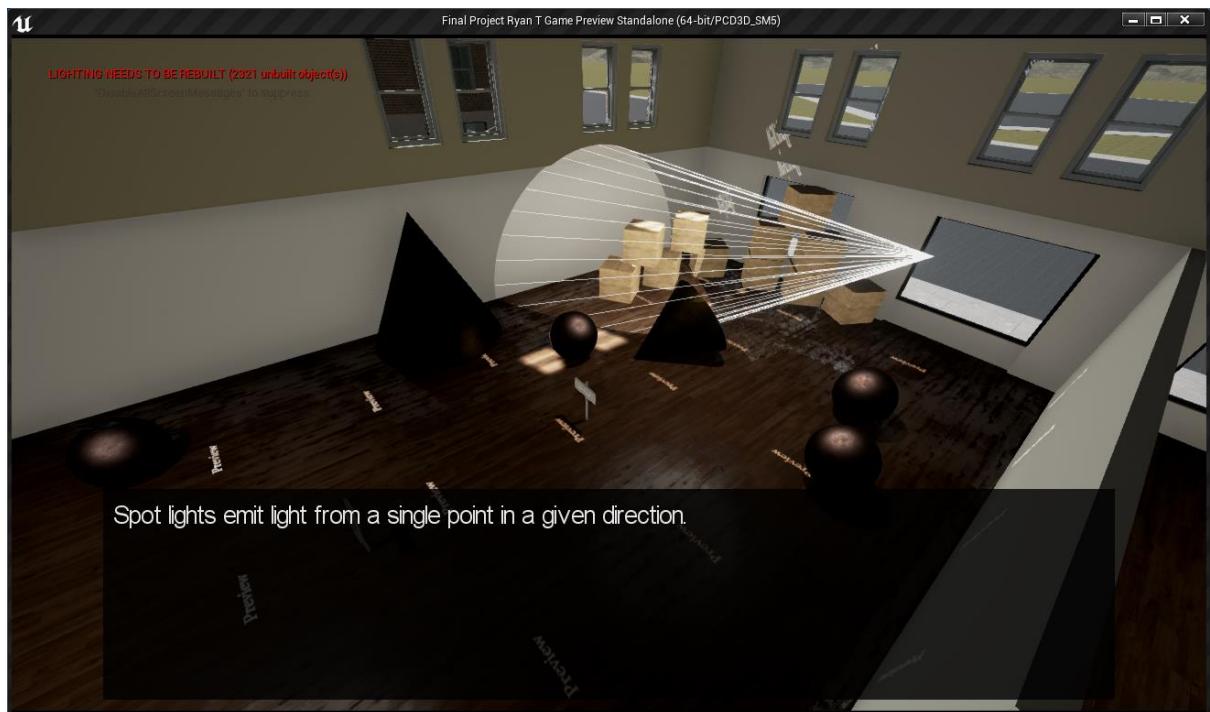


Figure 508

### Spot Light Sequence: Description of spots basic parameters.

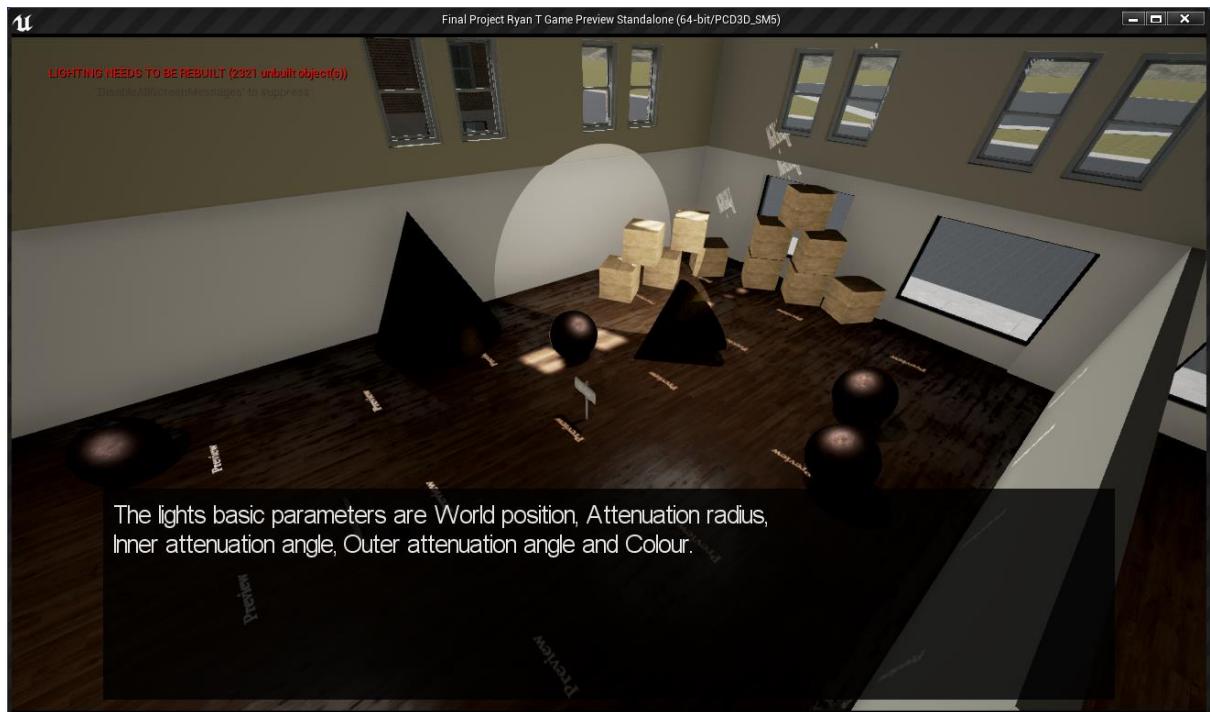


Figure 509

### Spot Light Sequence: Light move left and right to show position.

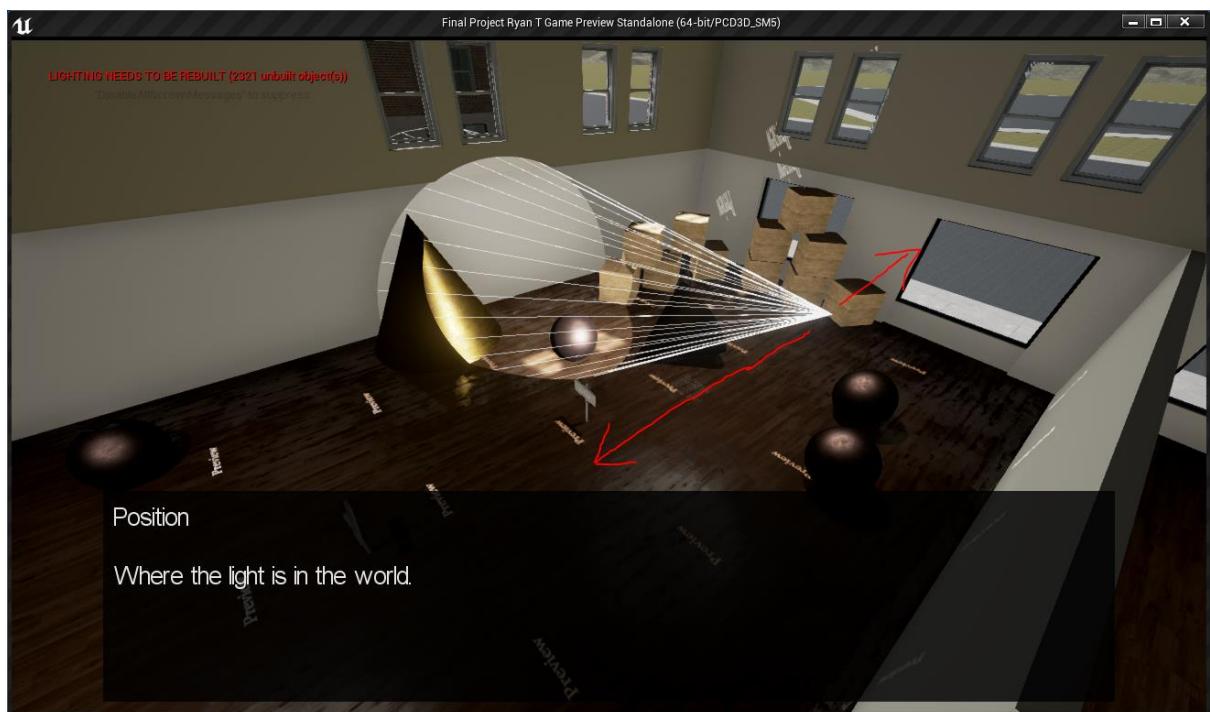


Figure 510

Spot Light Sequence: Debug draw of inner radius reduces in angle to demo attenuation.



Figure 511

Spot Light Sequence: Debug draw of outer radius reduces in angle to demo attenuation



Figure 512

Spot Light Sequence: Light radius shrinks and grows to show distance attenuation.

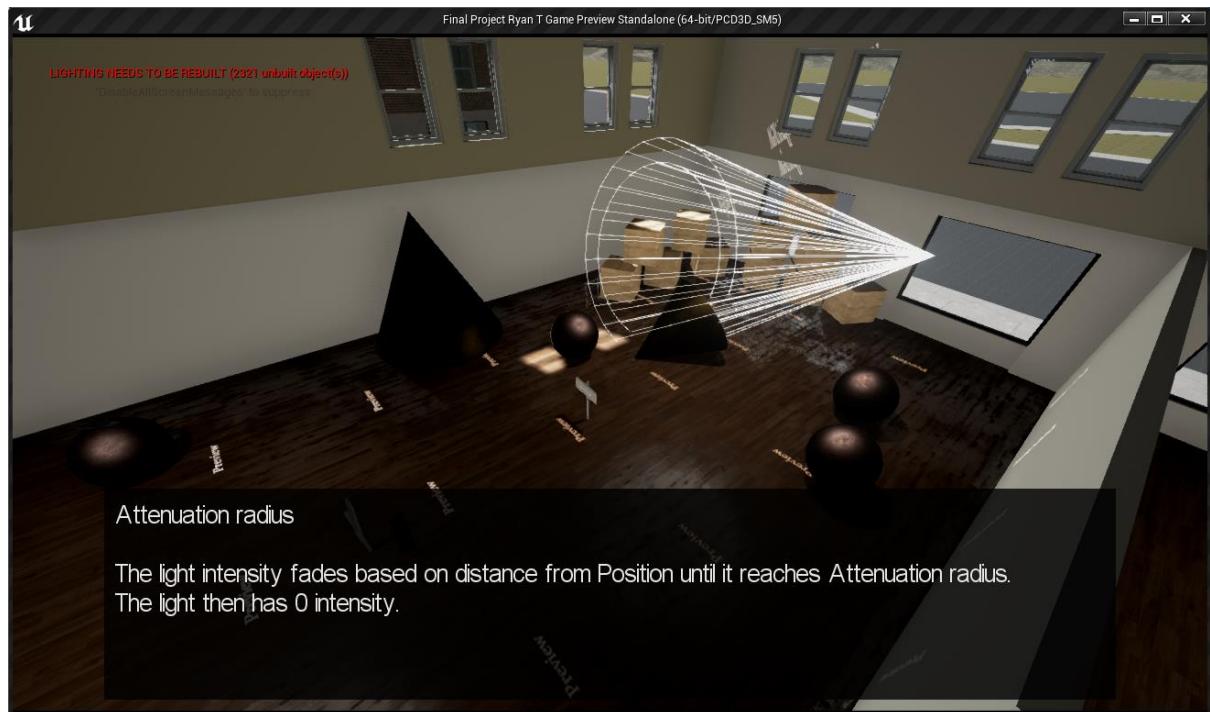


Figure 513

Spot Light Sequence: Light colour changes

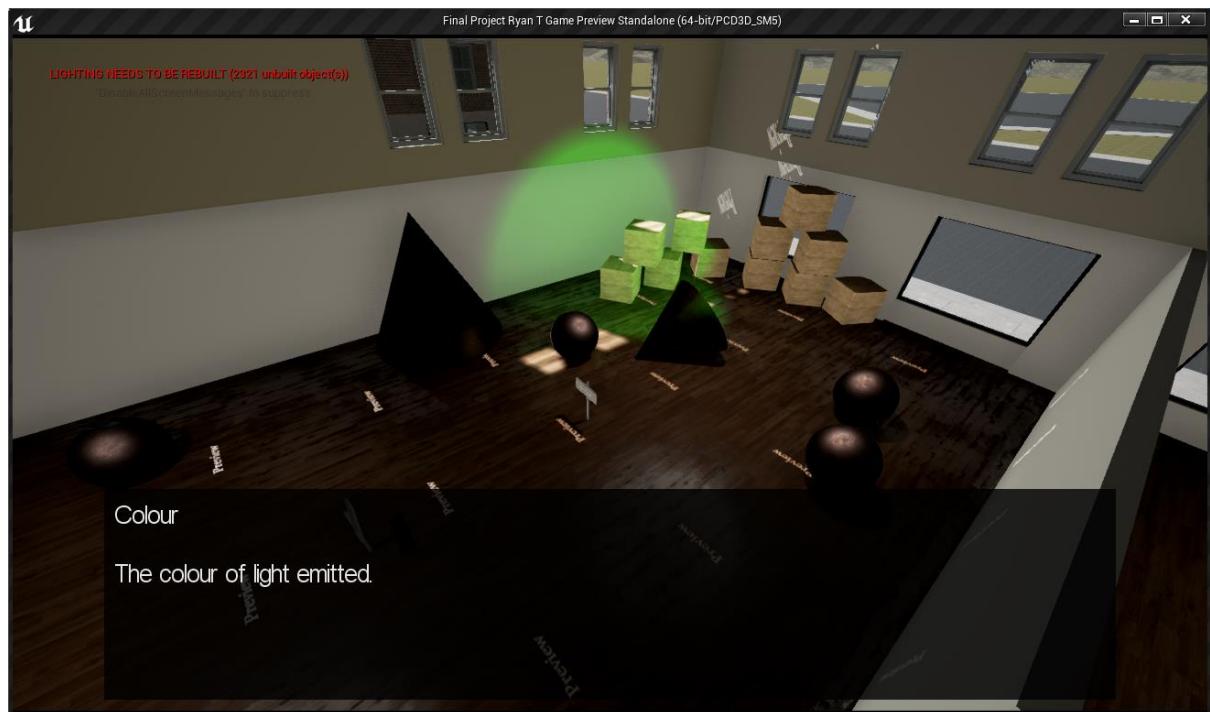


Figure 514

Spot Light Sequence: Shadows are enabled and light rotates from left to right to highlight shadows.

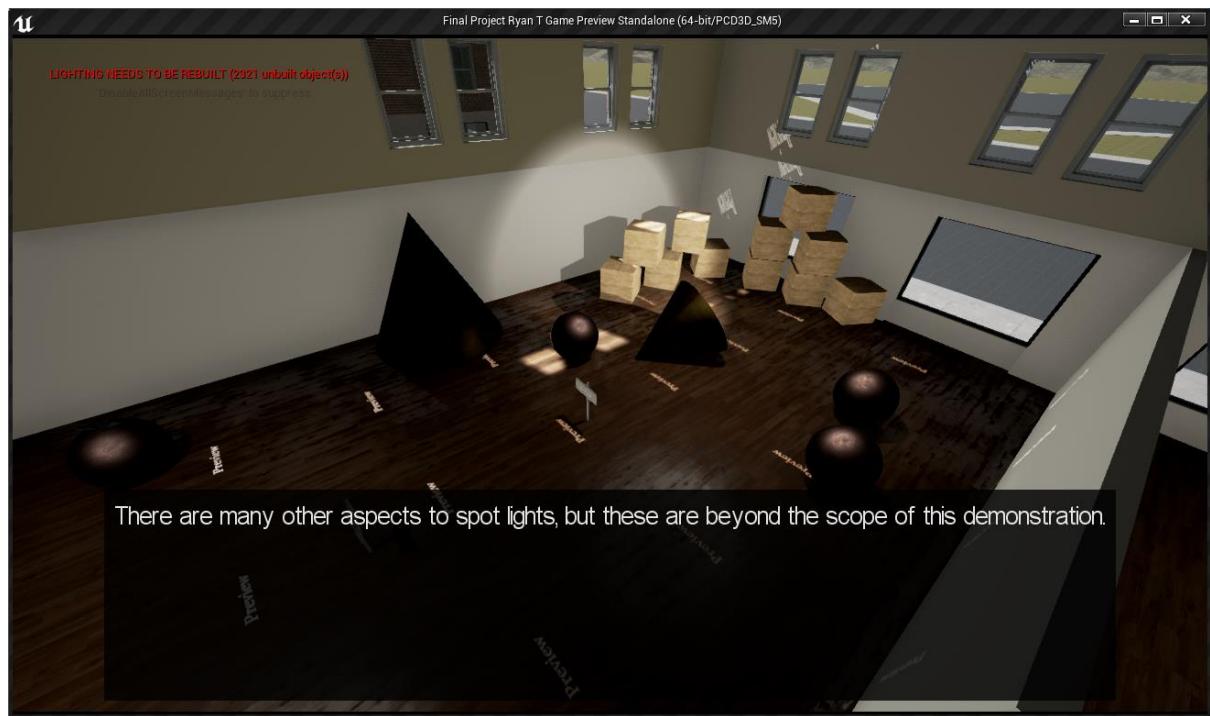


Figure 515 – Spotlight Movement (2 Images)

### Spot Light Sequence: Light fades back off and demo ends



Figure 516

#### 6.7.3. Directional Light Demo

Duplicated LightDemoActor and Camera from previous Point Light demo. This will be used as a base for the directional light demo. Text change or replaced.



Figure 517

Duplicated the whole of the point light level sequence as a base for the directional light demo.

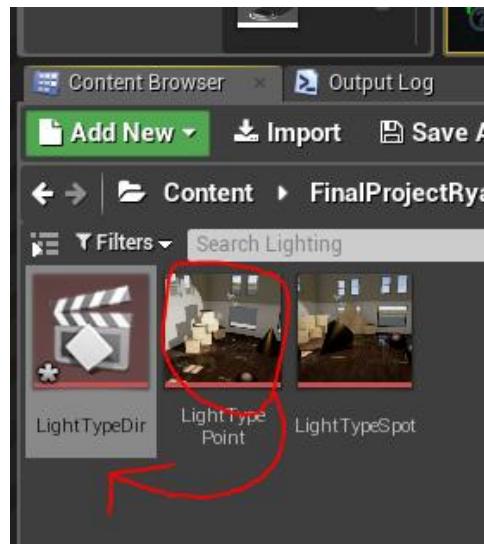


Figure 518

Changed the directional LightDemo actor to point to the newly duplicated level sequence.

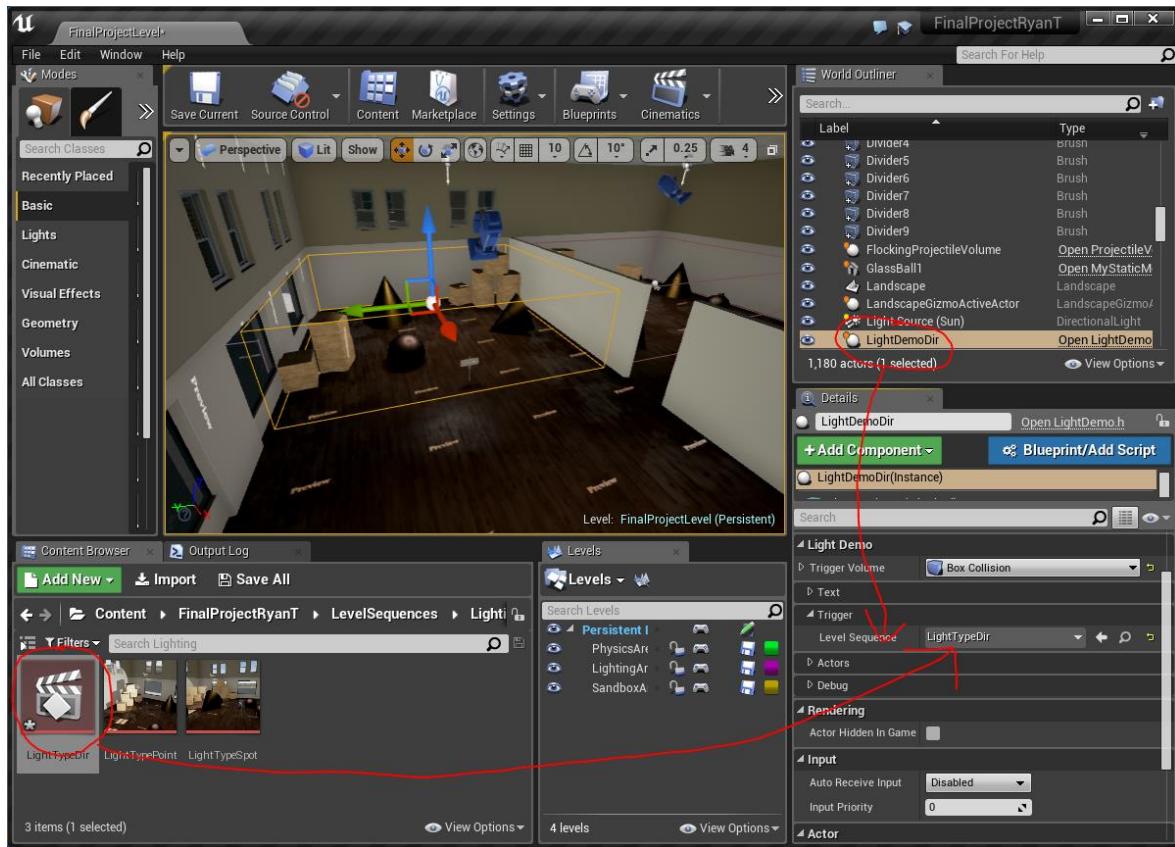


Figure 519

Added new directional light into scene for demo.

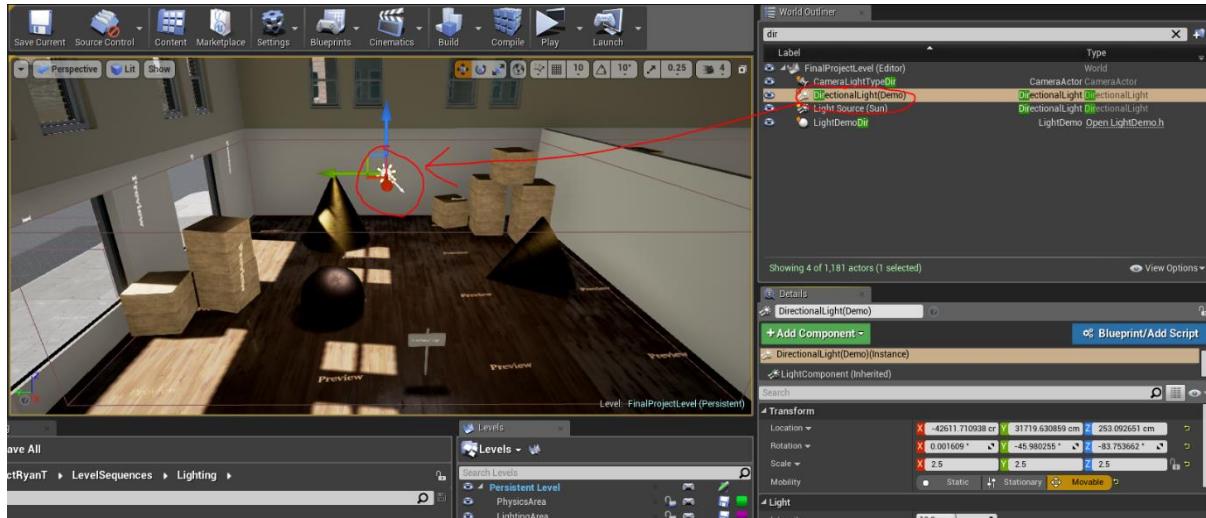


Figure 520

Extended Light Demo actor DebugDrawLight to draw directional light types (Just an arrow in direction of light)

```

145 //*****
146 // UpdatesubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
147 //*****
148 void ALightDemo::DebugDrawLight()
149 {
150     if (!Light)
151         return; // No light added so just return
152
153     ULightComponent* LightComp = Light->GetLightComponent();
154
155     /* Point Lights */
156     /*******/
157     if (LightComp->GetClass() == UPointLightComponent::StaticClass())
158     {
159         if (bShowLightRadius)
160         {
161             UPointLightComponent* Point = Cast<UPointLightComponent>(LightComp); // Cast from a light to a point light
162
163             // If it's a point light draw it's radius
164             DrawDebugSphere(GetWorld(), Point->GetComponentLocation(), Point->AttenuationRadius, 32, FColor::White);
165         }
166     }
167
168     /*******/
169     /* Spot Lights */
170     /*******/
171     if (LightComp->GetClass() == USpotLightComponent::StaticClass())
172     {
173         if (bShowLightRadius)
174         {
175             USpotLightComponent* Spot = Cast<USpotLightComponent>(LightComp); // Cast from a light to a spot light
176
177             // If it's a spot light draw it's angled cones
178             float OutAngle = FMath::DegreesToRadians(Spot->OuterConeAngle);
179             float InAngle = FMath::DegreesToRadians(Spot->InnerConeAngle);
180
181             DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X), Spot->AttenuationRadius, OutAngle, OutAngle, 32, FColor::White);
182             DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X), Spot->AttenuationRadius, InAngle, InAngle, 32, FColor::White);
183         }
184     }
185
186     /*******/
187     /* Directional Lights */
188     /*******/
189     if (LightComp->GetClass() == UDirectionalLightComponent::StaticClass())
190     {
191         if (bShowLightRadius)
192         {
193             UDirectionalLightComponent* DirLight = Cast<UDirectionalLightComponent>(LightComp); // Cast from a light to a directional light
194             FVector ArrowEnd = DirLight->GetComponentLocation() + DirLight->GetDirection() * 180.0f; // The arrow will be 180 world units long
195             DrawDebugDirectionalArrow(GetWorld(), DirLight->GetComponentLocation(), ArrowEnd, 40.0f, FColor::White, false, 0.0f, 0, 3.0f);
196         }
197     }
198 }
199

```

Figure 521

Completed Level Sequence for directional lights.

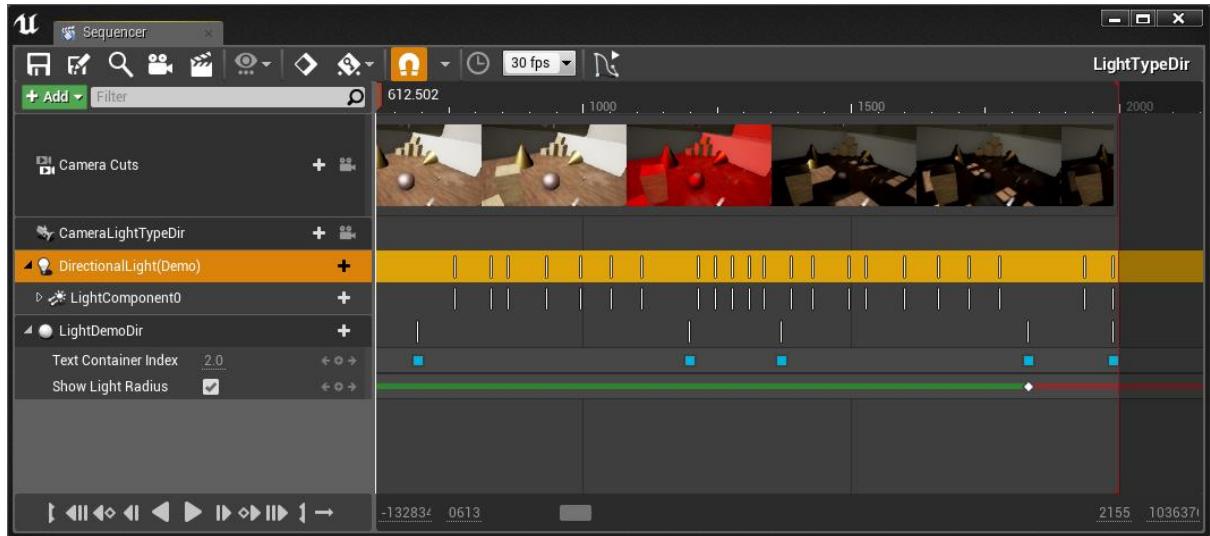


Figure 522

Directional Light Sequence: Camera cuts to a suitable position

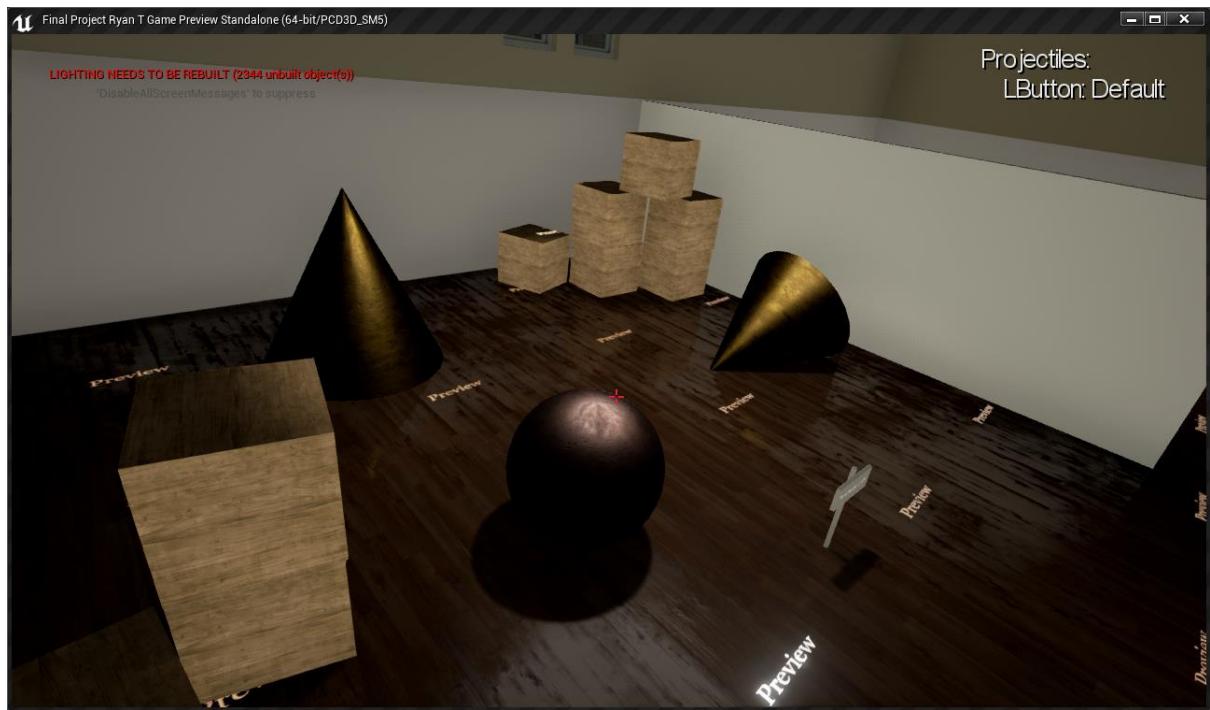


Figure 523

### Directional Light Sequence: Light fades up

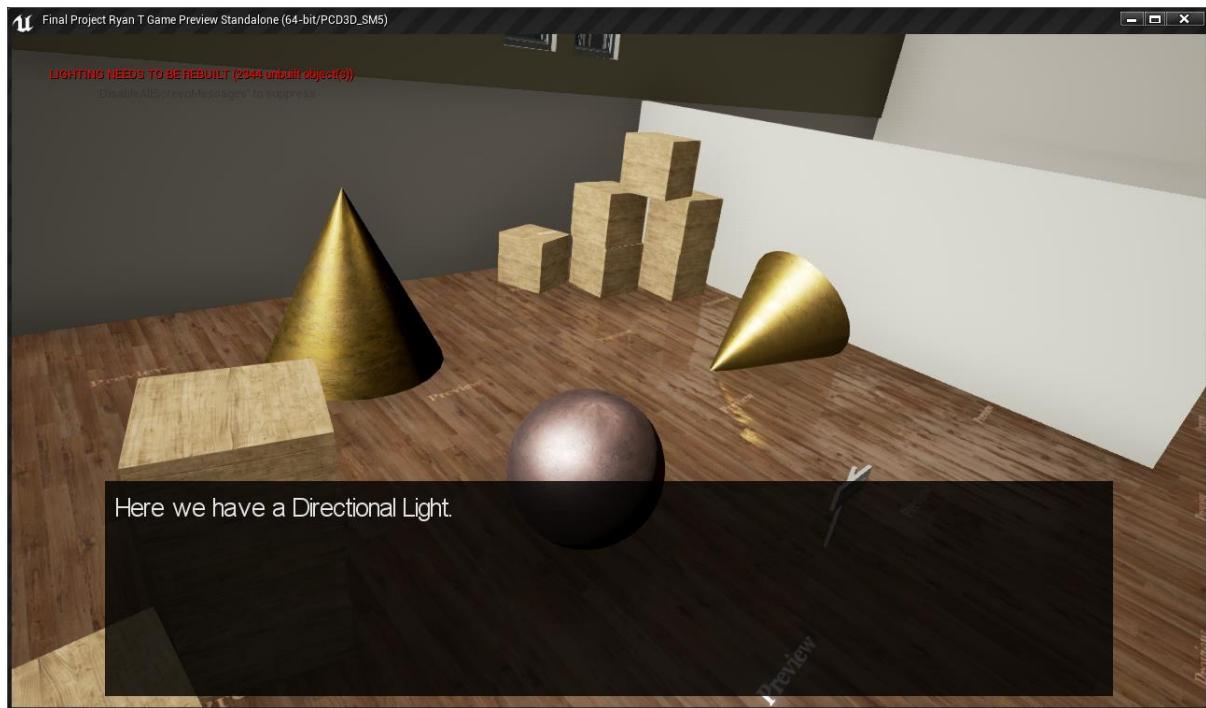
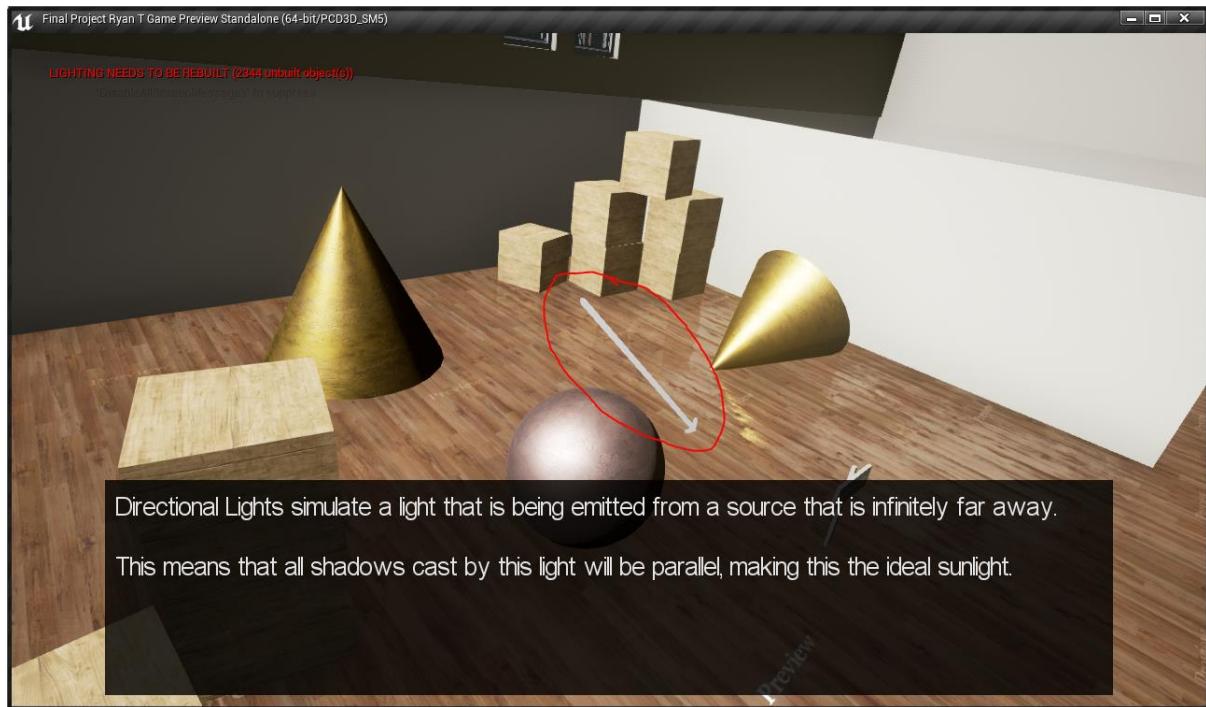


Figure 524

### Directional Light Sequence: Debug arrow is show in light direction



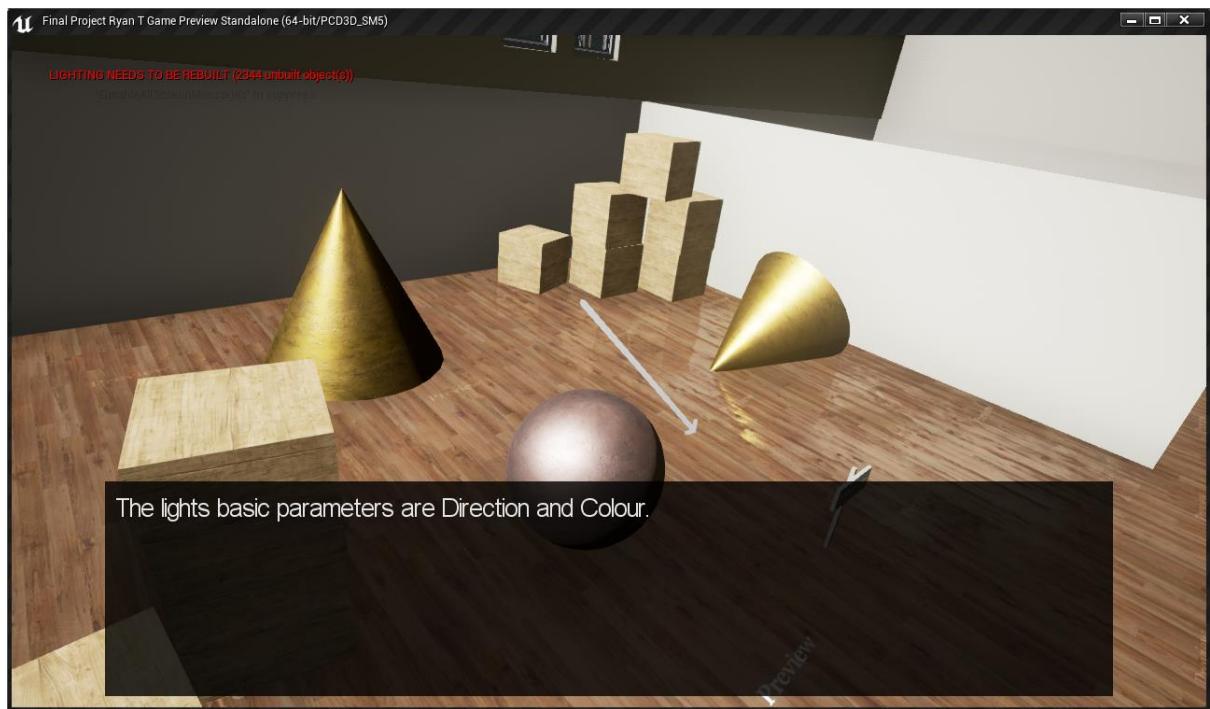


Figure 525

Directional Light Sequence: Light direction is rotated arrow as light moves

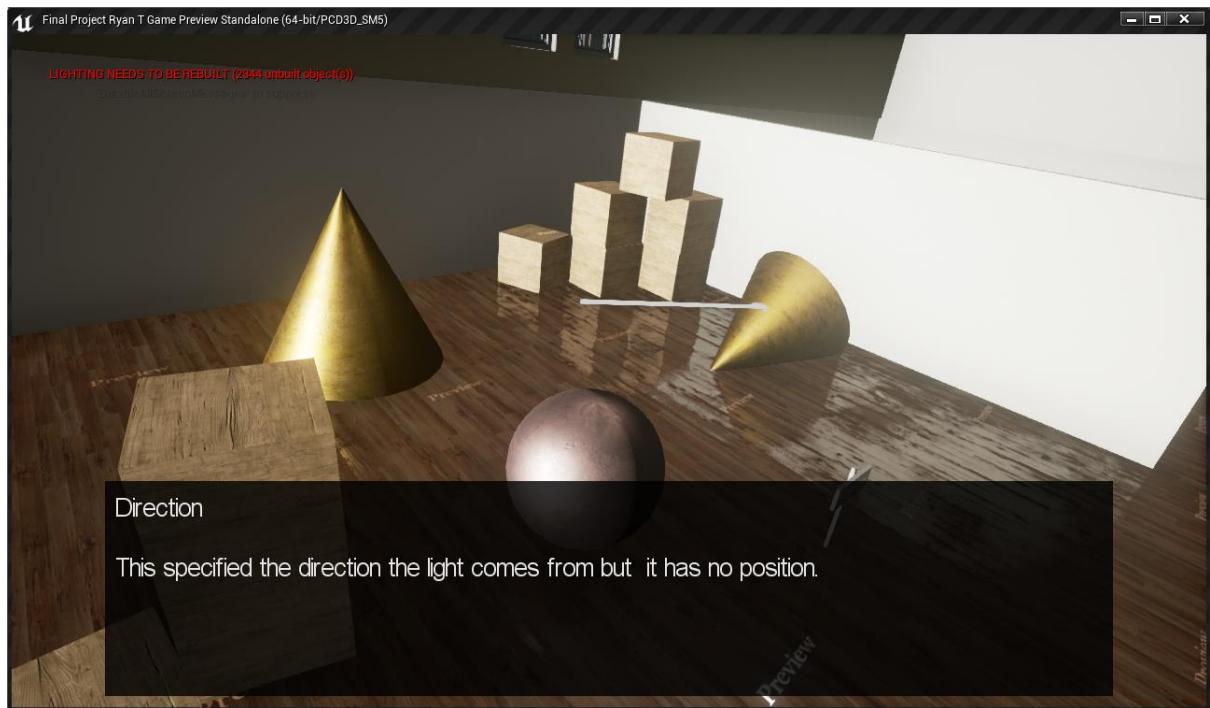


Figure 526

### Directional Light Sequence: Light colour is changed

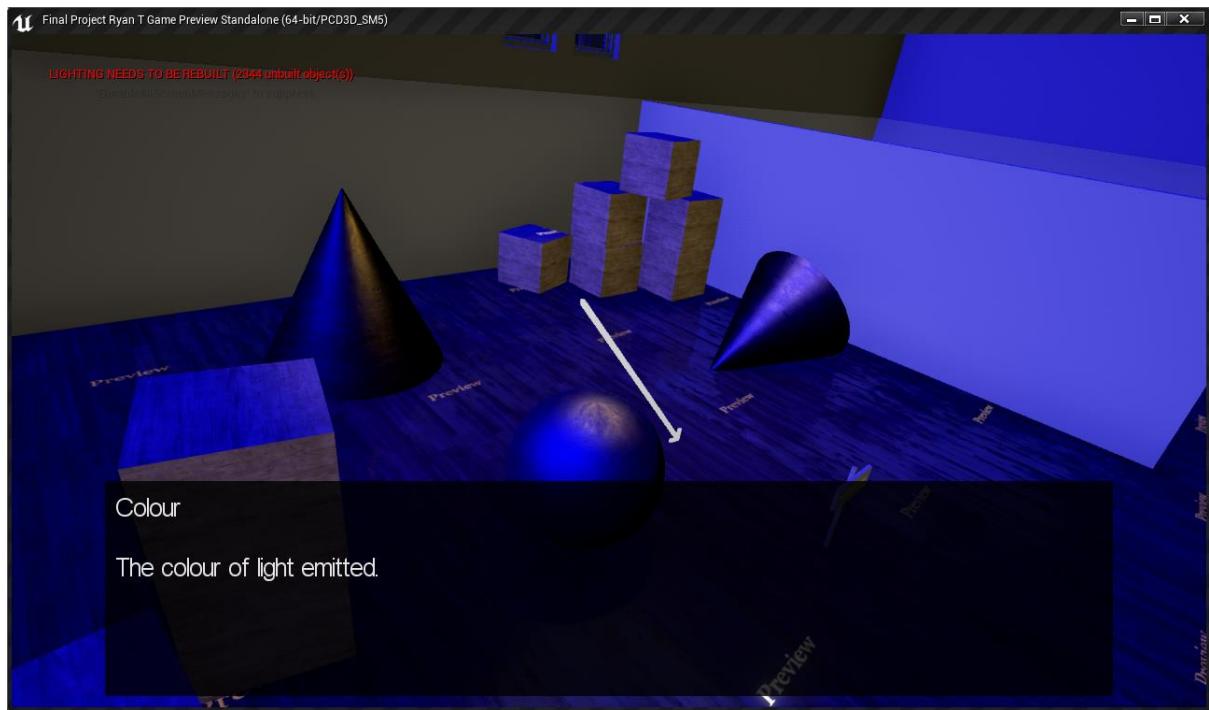


Figure 527

Directional Light Sequence: Shadows are turned on and Light direction is rotated to highlight them. Arrow also moves.

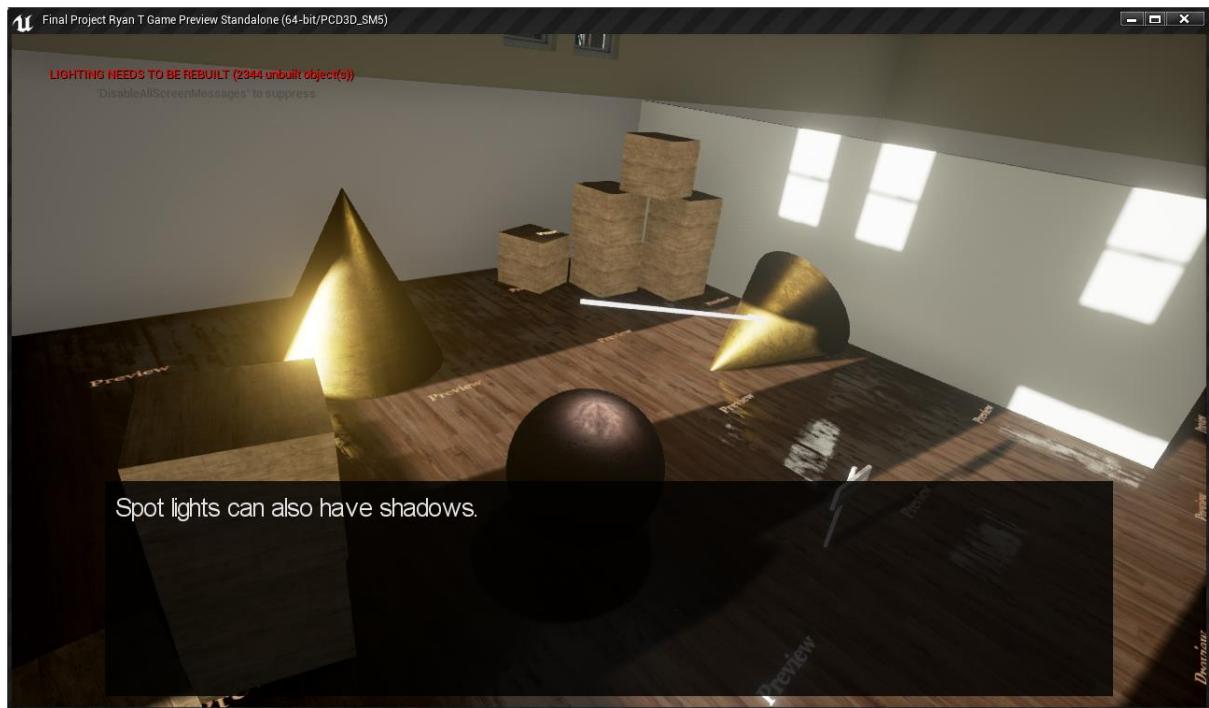




Figure 528 – Toggling Shadows (2 Images)

Directional Light Sequence: Light fades back down, text goes off, demo ends



Figure 529

## 6.8. Post Processing Effects Building (Lighting Area)

Added 6 LightDemo Actors and 6 cameras to PostProcessing Room ready for sequencing.



Figure 530

Created 6 empty levels sequences ready for sequencing.

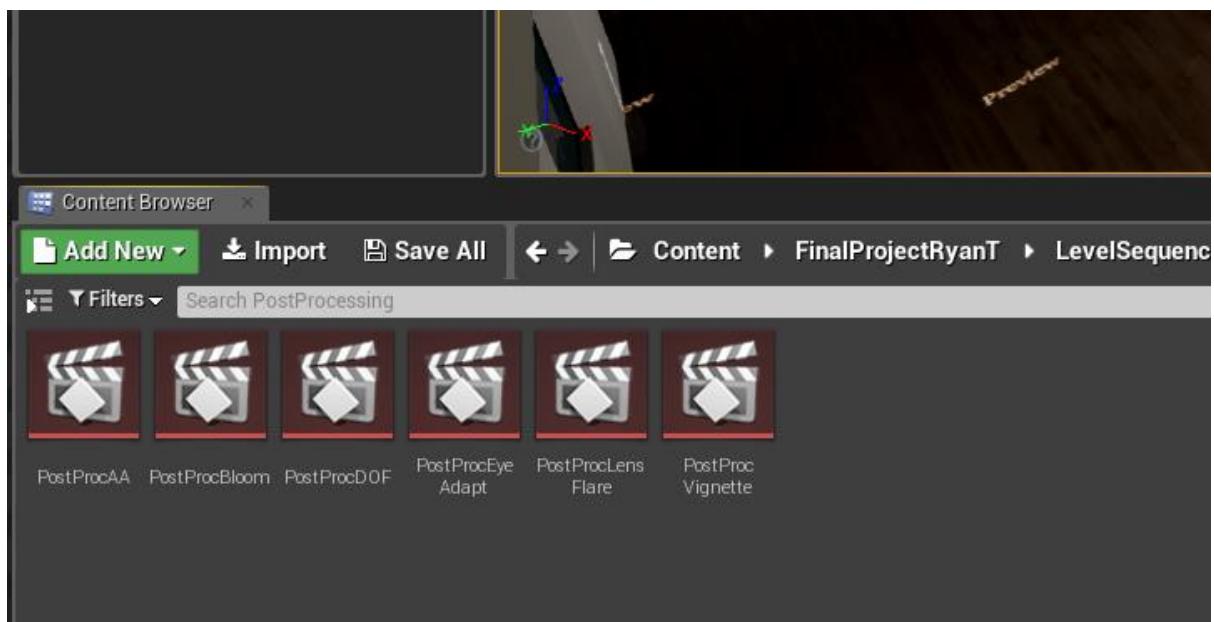


Figure 531

Linked all 6 LightDemo actors to their relevant camera and level sequence.

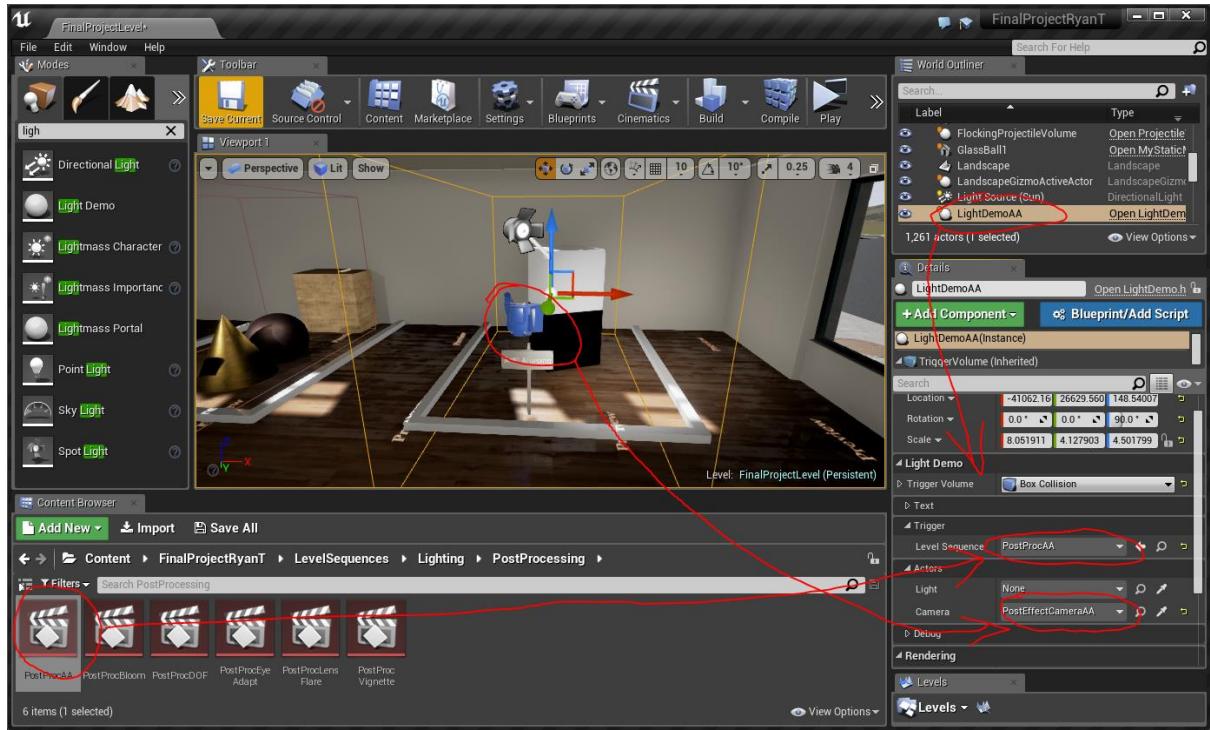


Figure 532

Added new function to LightDemo actor to set AA mode. Also added UPROPERTY AntiAliasMode to allow setting from timeline.

```

44
45     UPROPERTY()
46     ULevelSequencePlayer* SequencePlayer = nullptr;
47
48     UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|Debug")
49     bool bShowLightRadius = false;
50
51     UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|PostEffects")
52     float AntiAliasMode = 2; // Has to be a float not int to use UPROPERTY Interp. Otherwise it won't appear in timeline
53
54     UFUNCTION()
55     void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
56     UFUNCTION()
57     void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex);
58
59     private:
60     void OnEndSequenceEvent();
61     void UpdateSubtitleText();
62     void DebugDrawLight();
63     void SetAAMode(int Mode);
64
65 };

```

Figure 533

### New SetAAMode function (CPP file)

```
// ****
// SetAAMode - Updates AntiAlias mode
// ****
void ALightDemo::SetAAMode(int Mode)
{
    // run a console command to change the AA mode

    // 0 : off
    // 1 : FXAA
    // 2 : TemporalAA
    // 3 : MSAA

    FString Command = "r.DefaultFeature.AntiAliasing ";
    Command.AppendInt(Mode);
    GetWorld()->GetFirstPlayerController()->ConsoleCommand(Command, false);
}
```

Figure 534

#### 6.8.1. Anti-Aliasing Demo

##### Sequenced AA

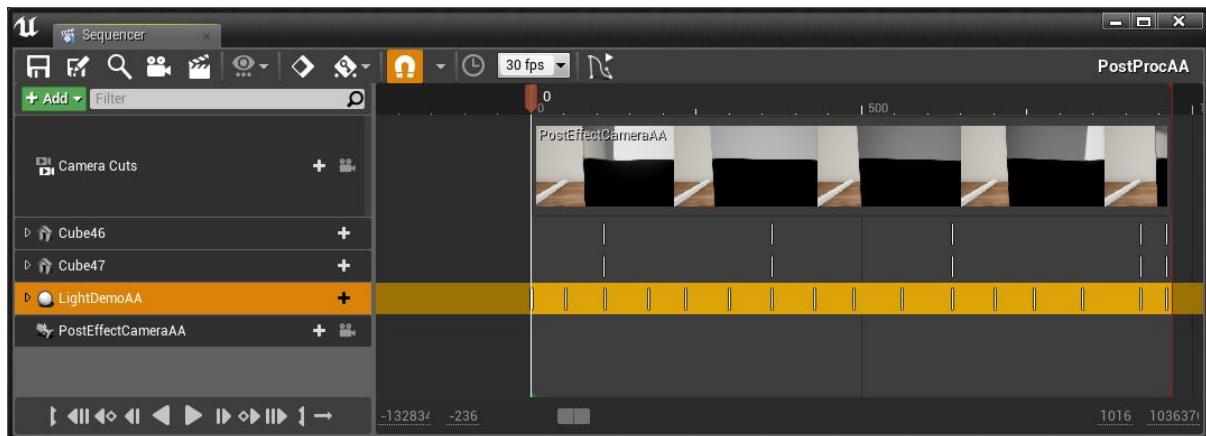


Figure 535

### AA Sequence: (Intro and camera cut, cubes start spinning)

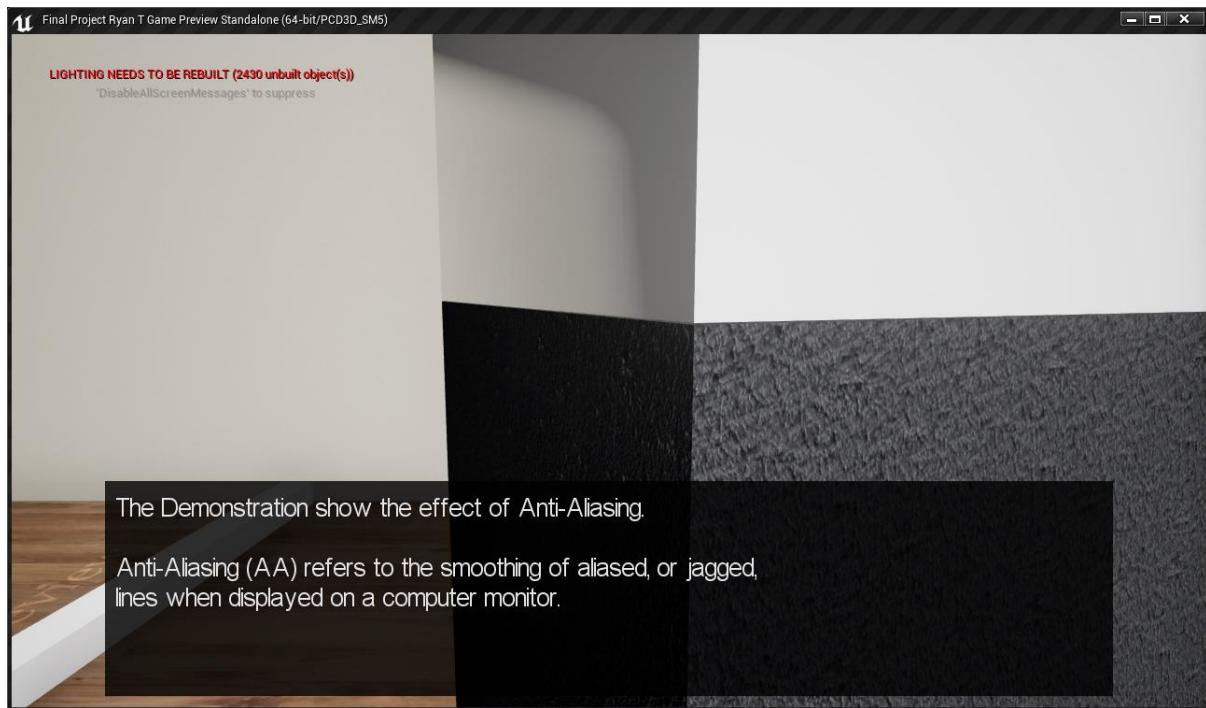


Figure 536

### AA Sequence: AA ON

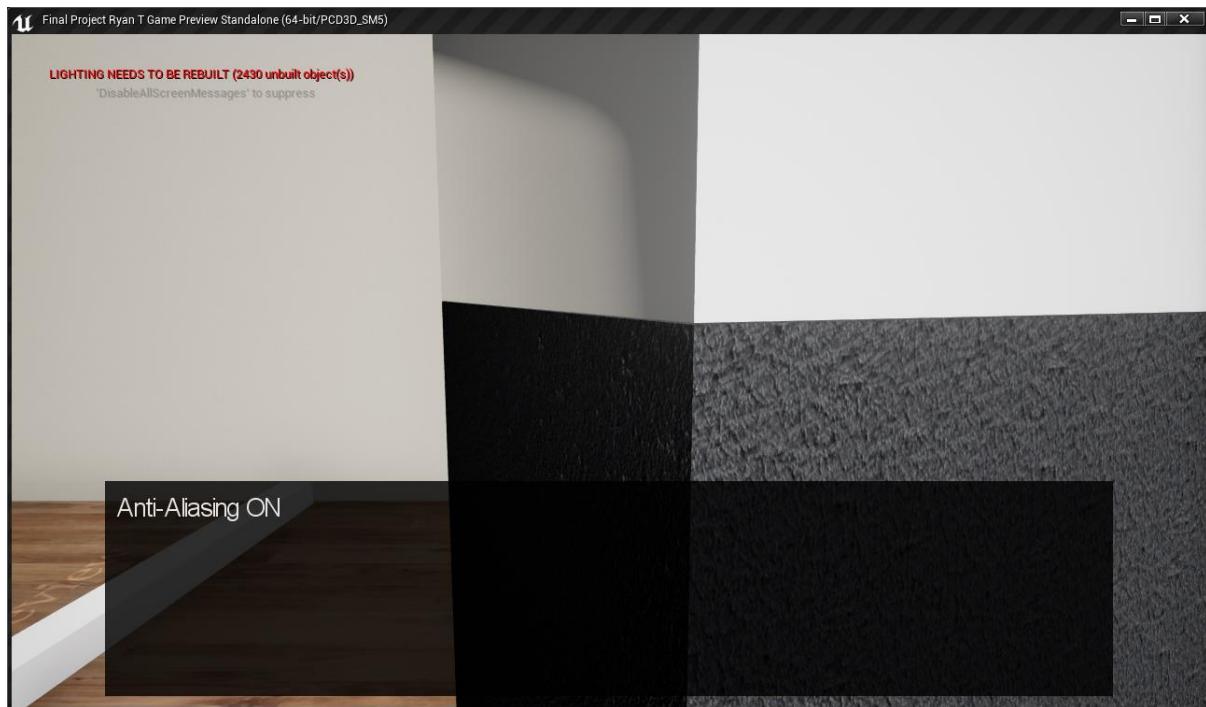


Figure 537

## AA Sequence: AA OFF

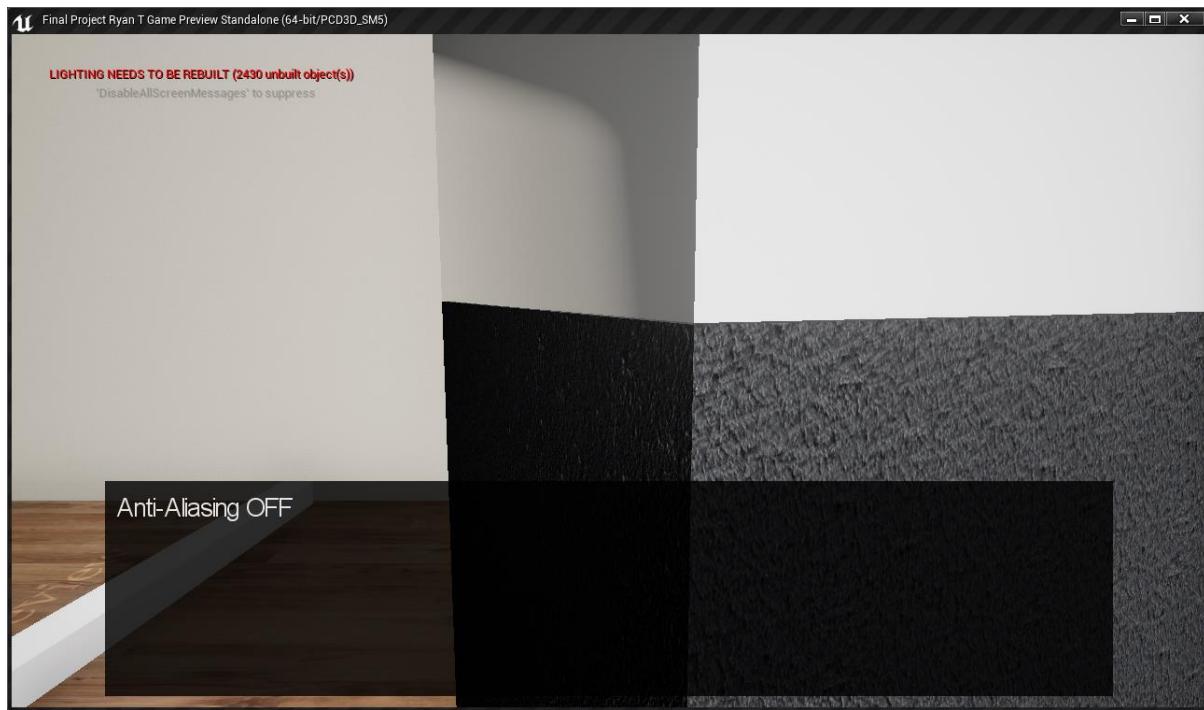


Figure 538

### 6.8.2. Depth of Field Demo

#### Sequenced DOF

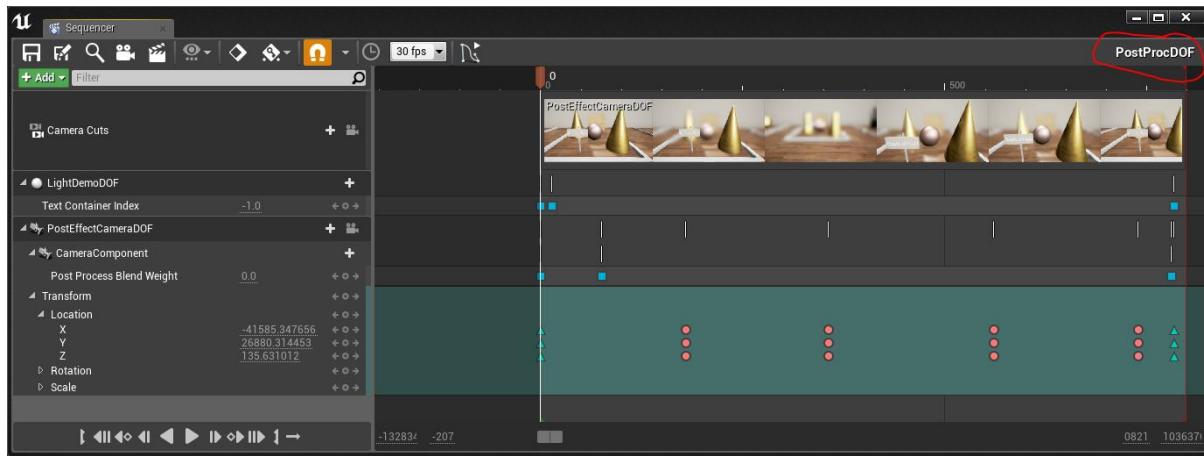


Figure 539

### DOF Sequence: Effect off, display text

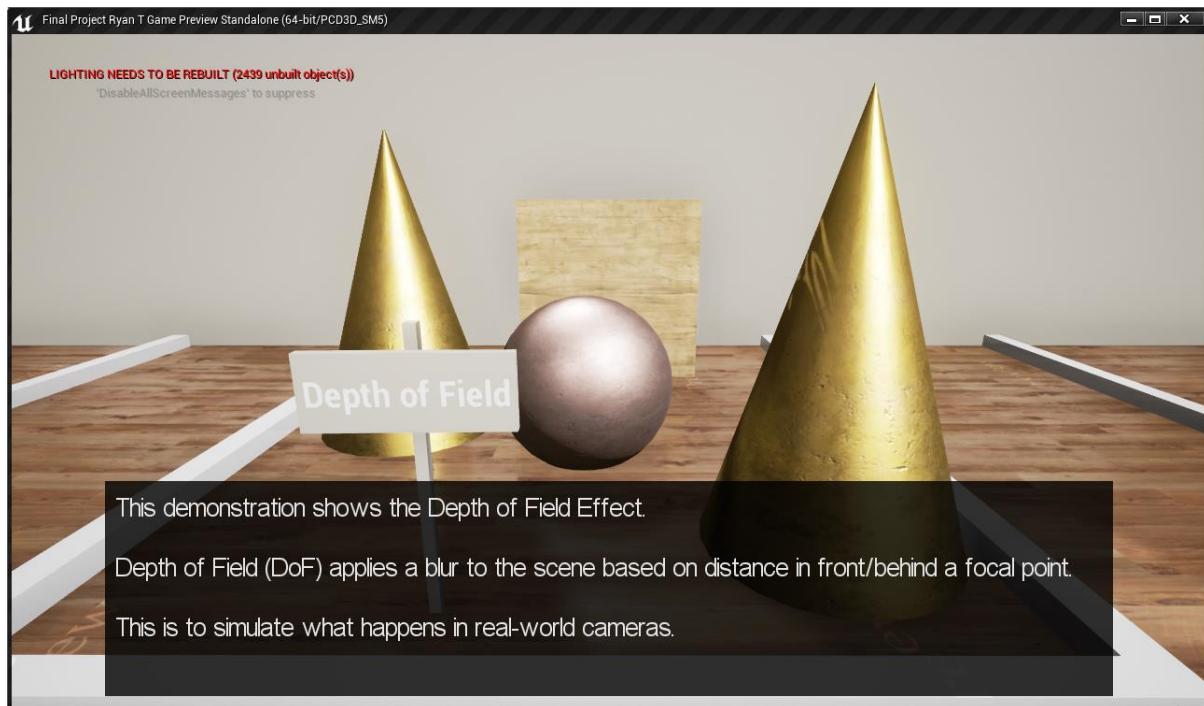


Figure 540

### DOF Sequence: Effect ON

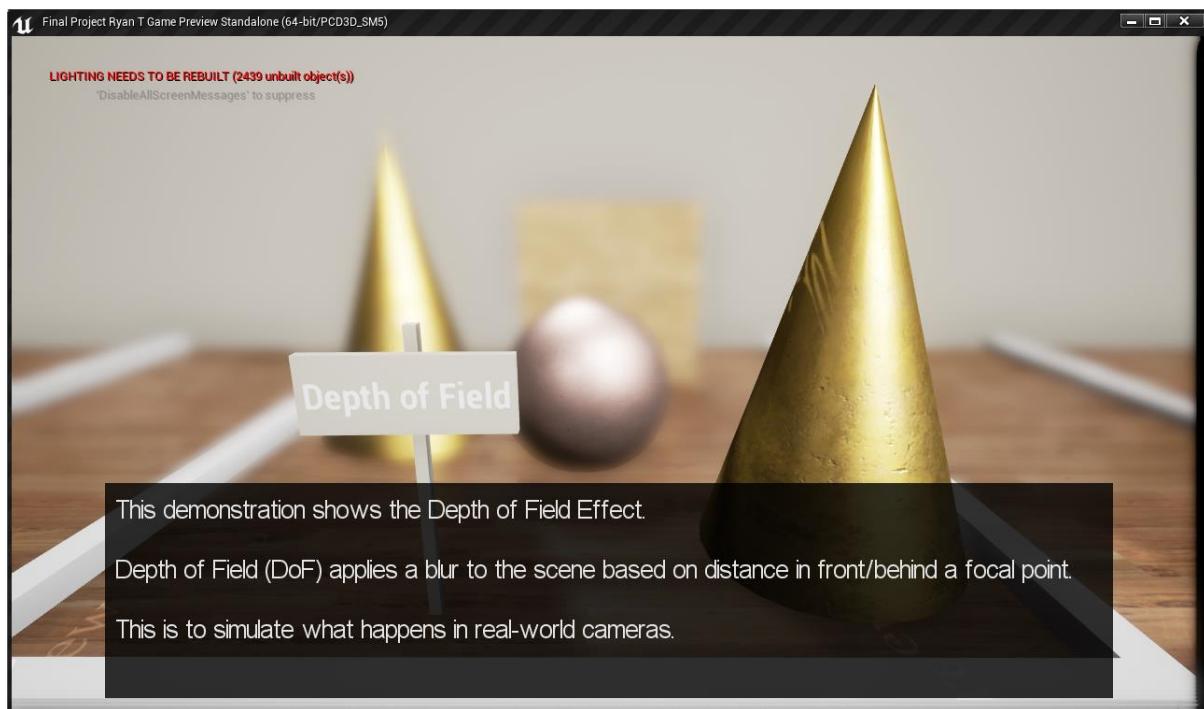


Figure 541

DOF Sequence: Camera slides back highlighting the focal point.

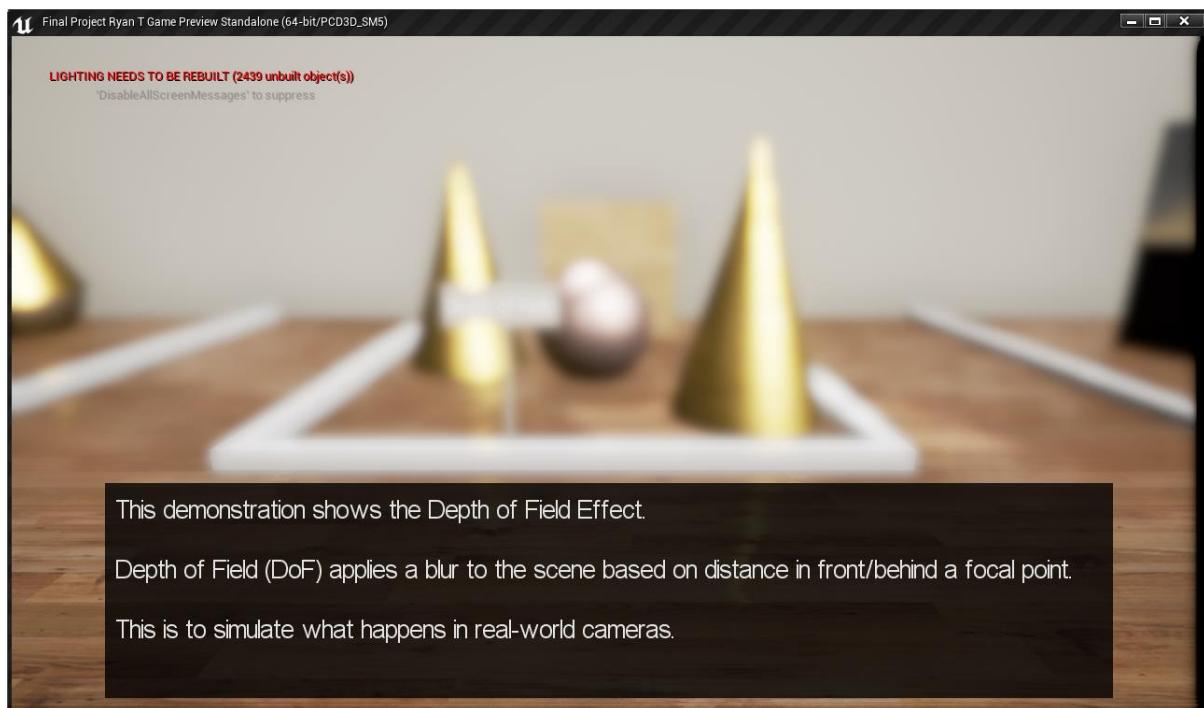


Figure 542

DOF Sequence: Camera slides forwards highlighting the focal point

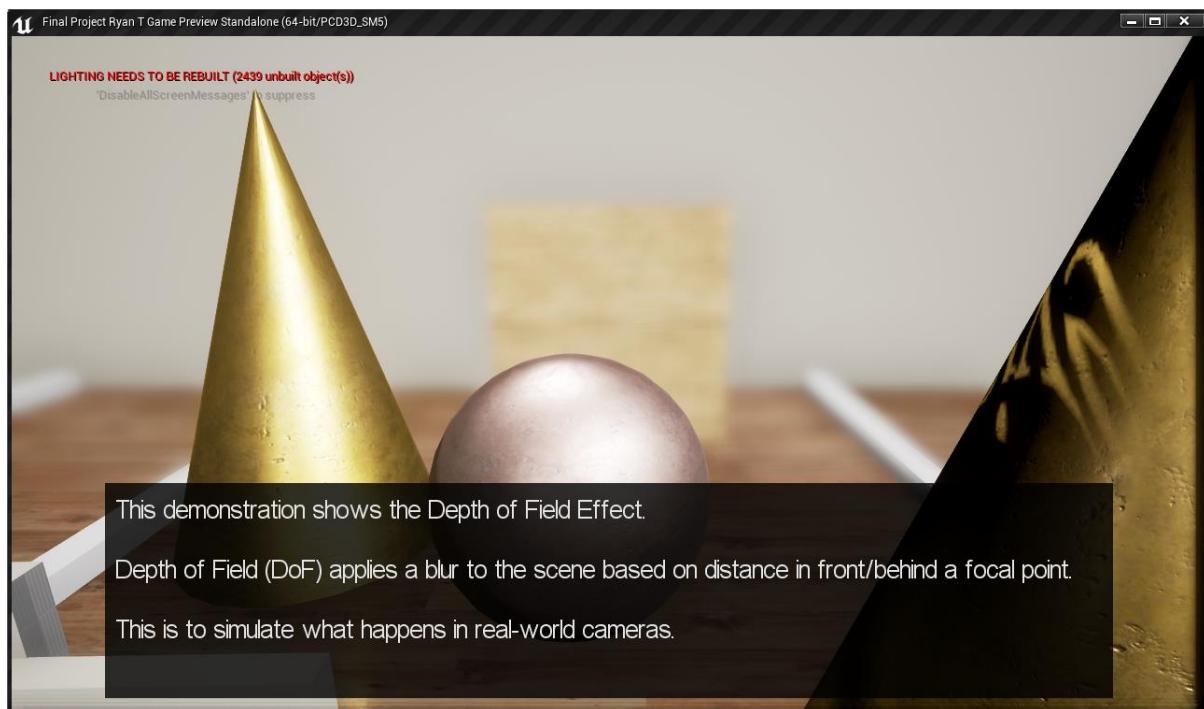


Figure 543

## DOF Sequence: Effect Off, Text Off, End Sequence

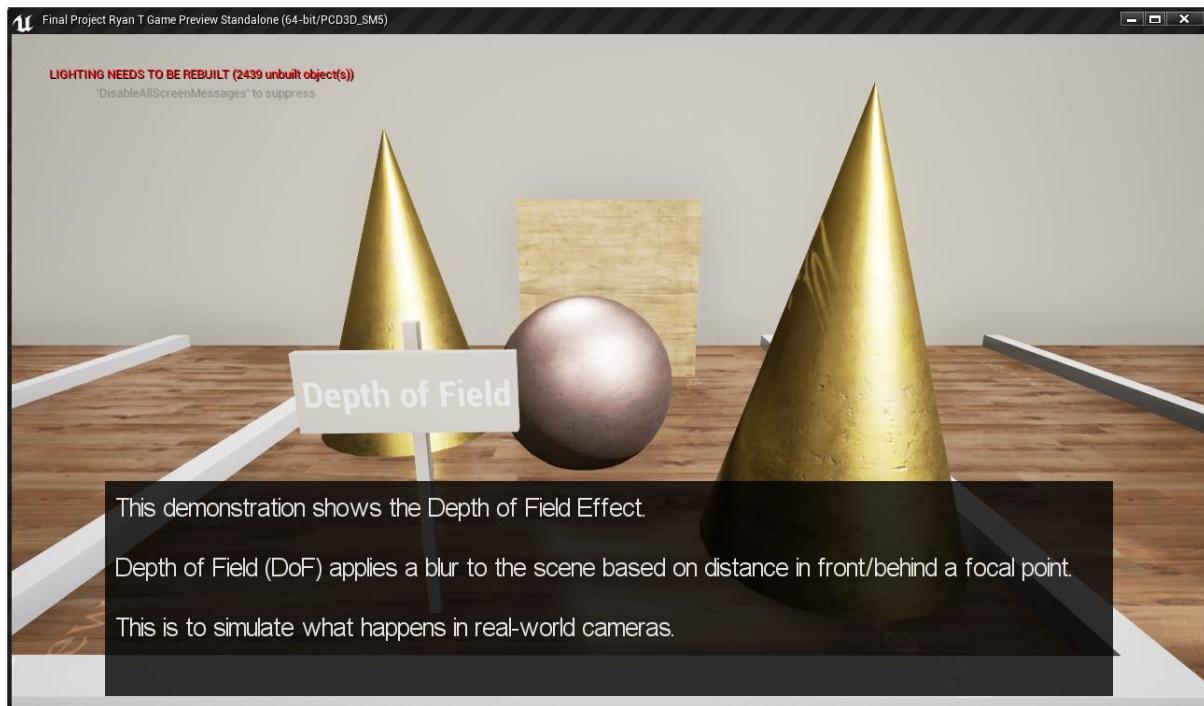


Figure 544

### 6.8.3. Vignette Demo

#### Sequenced Vignette

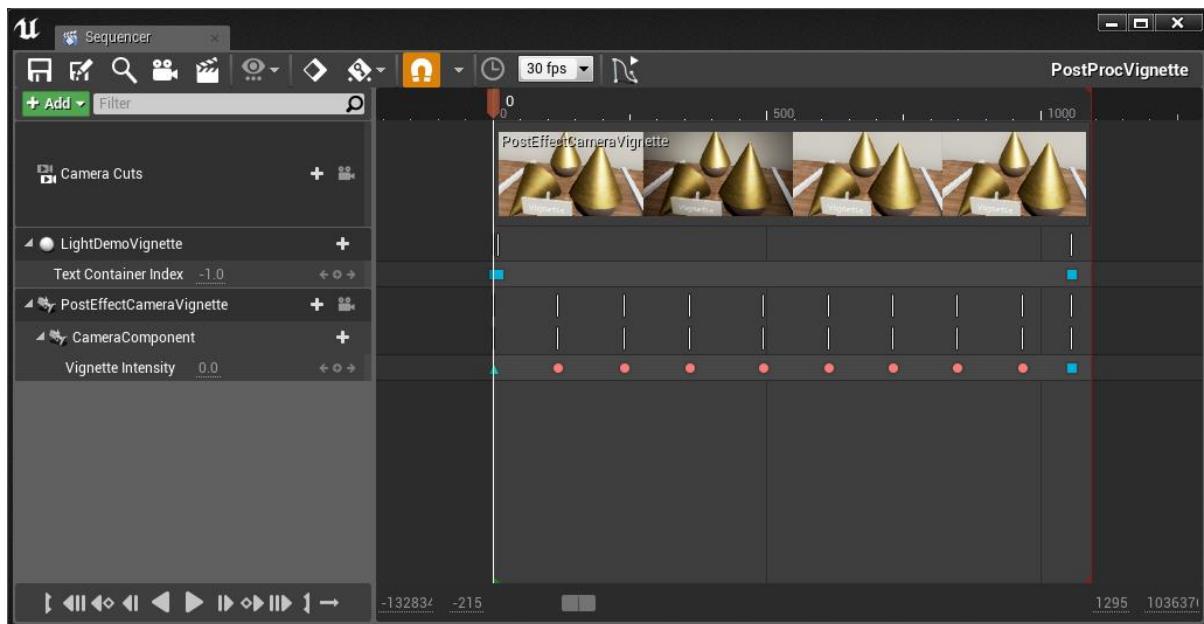


Figure 545

### Vignette Sequence: Show Text

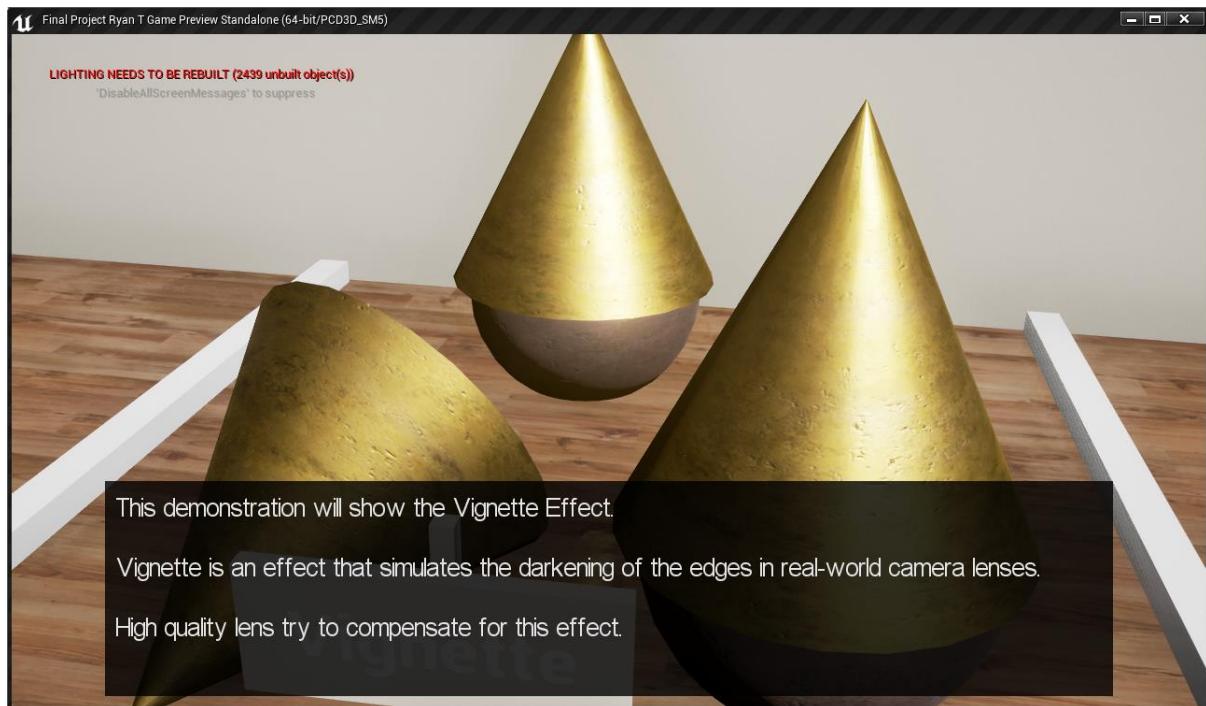


Figure 546

### Vignette Sequence: Vignete effect fades up and down to demonstrate

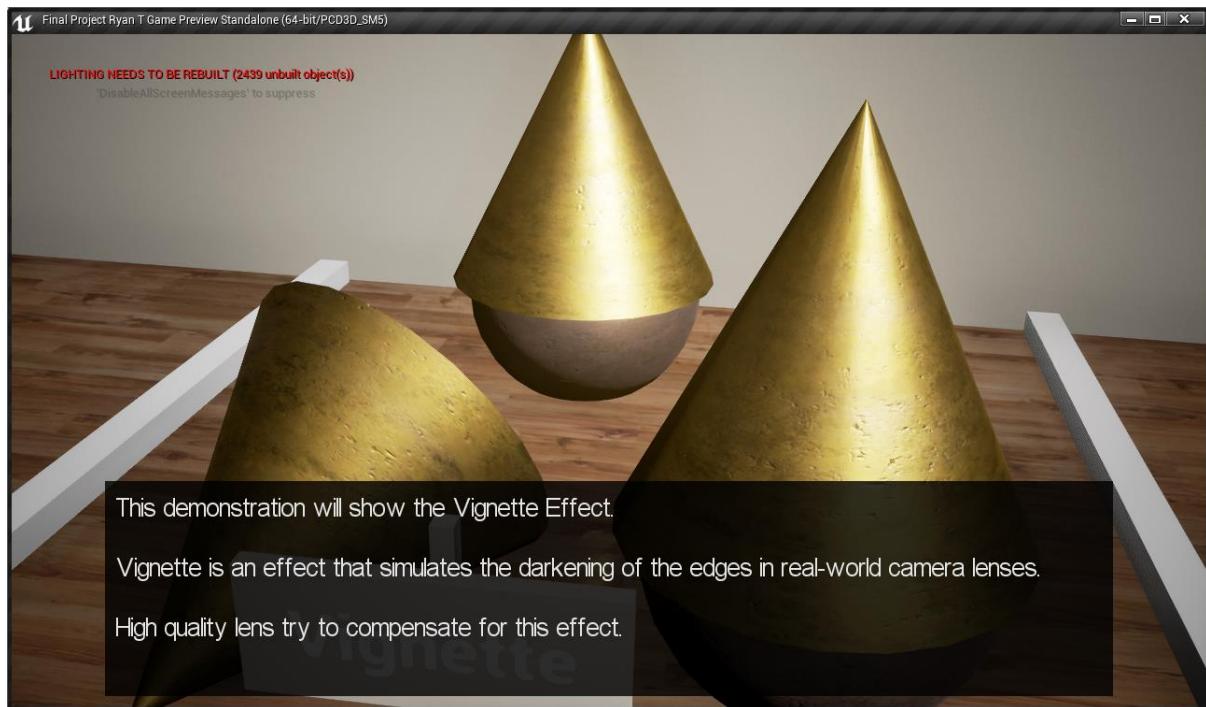


Figure 547

#### 6.8.4. Lens Flare Demo

##### Sequenced Lens Flare

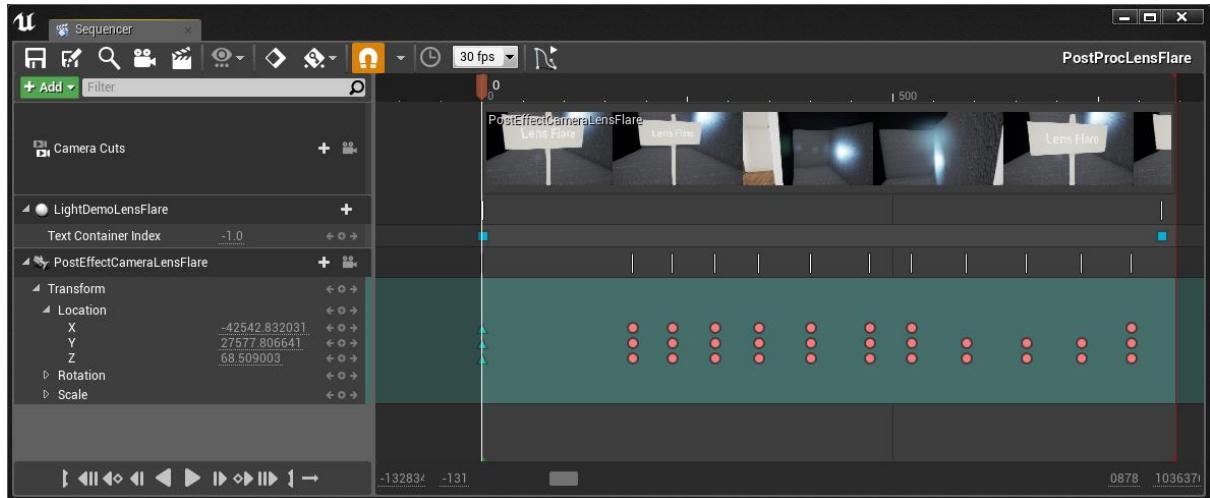


Figure 548

##### Lens Flare Sequence: Display Text and cut camera



Figure 549

Lens Flare Sequence: Camera moved around the opening to the container highlighting the lens flare effect

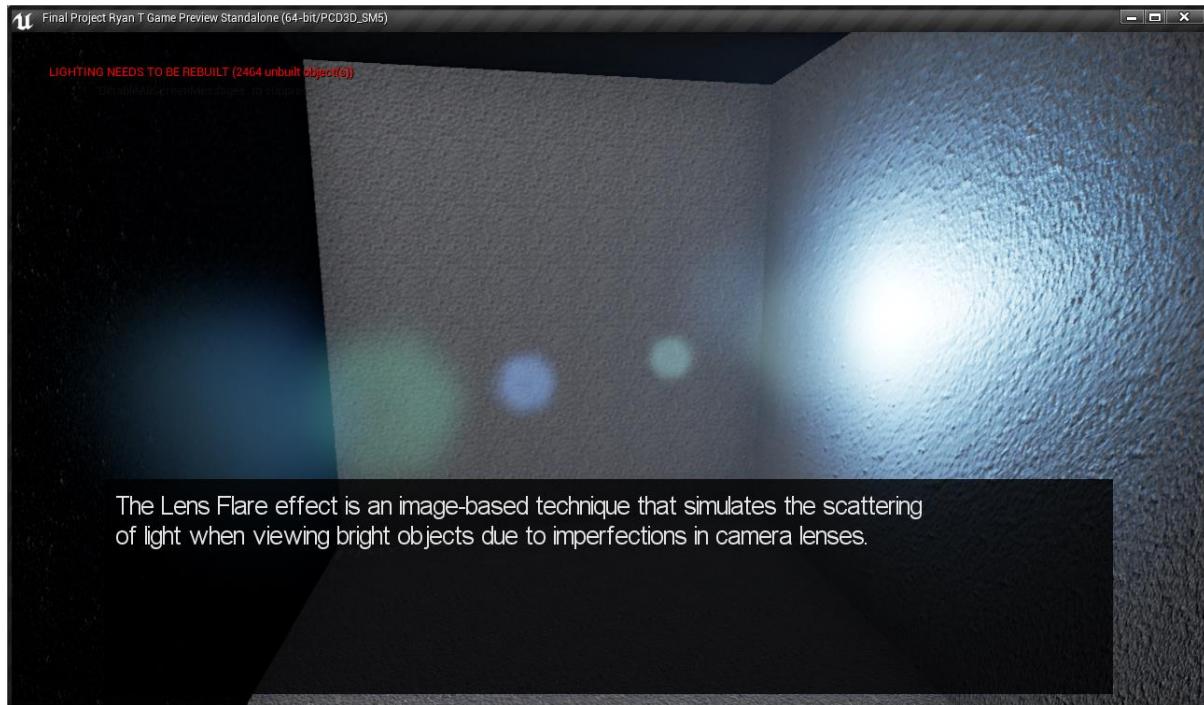


Figure 550

#### 6.8.5. Bloom Demo

##### Sequenced Bloom

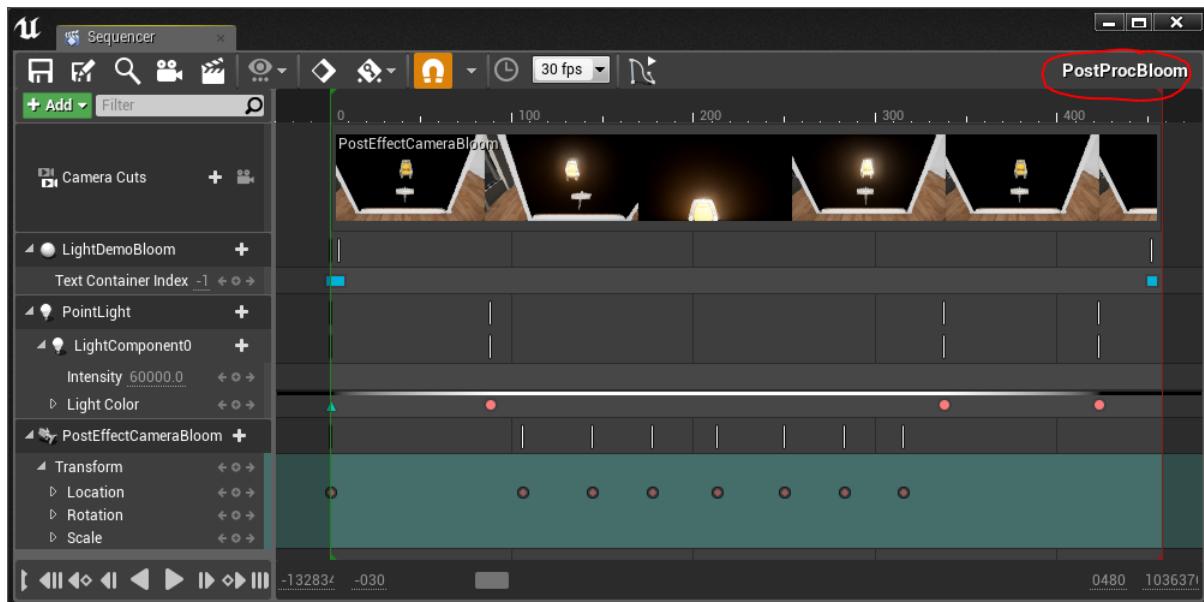


Figure 551

### Bloom Sequence: Cam Cuts, Text Displayed, Effect fades up

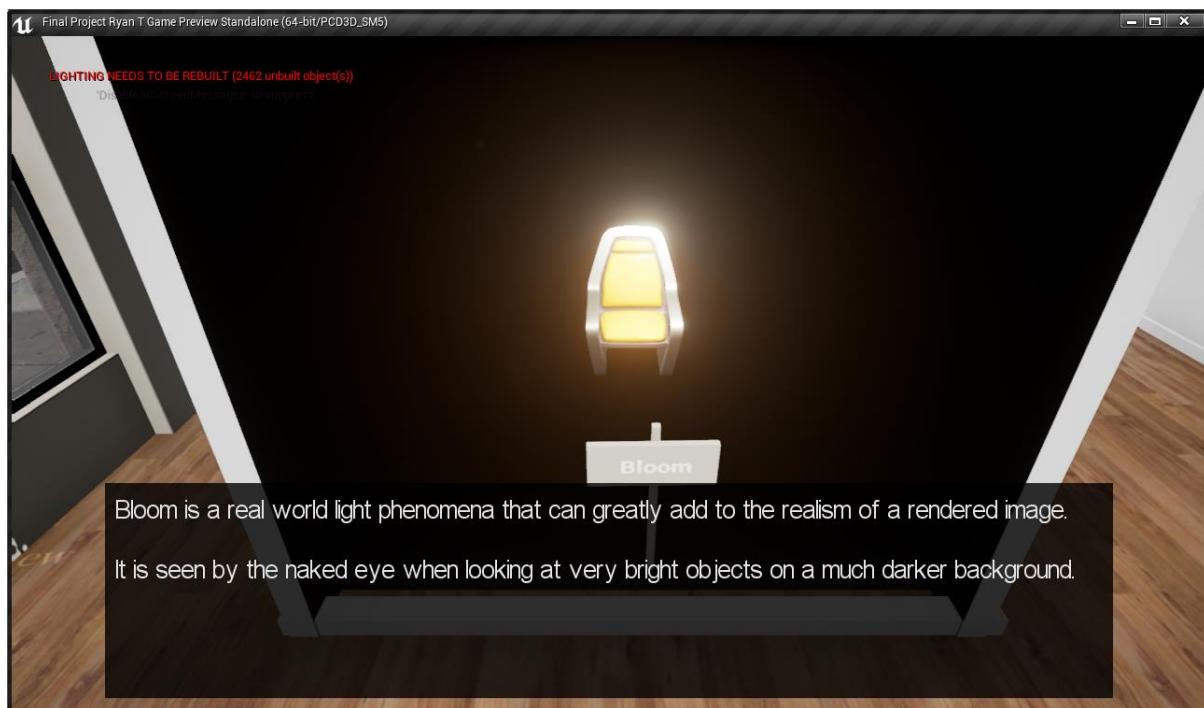


Figure 552

### Bloom Sequence: Cam moves into container around chair then back out again

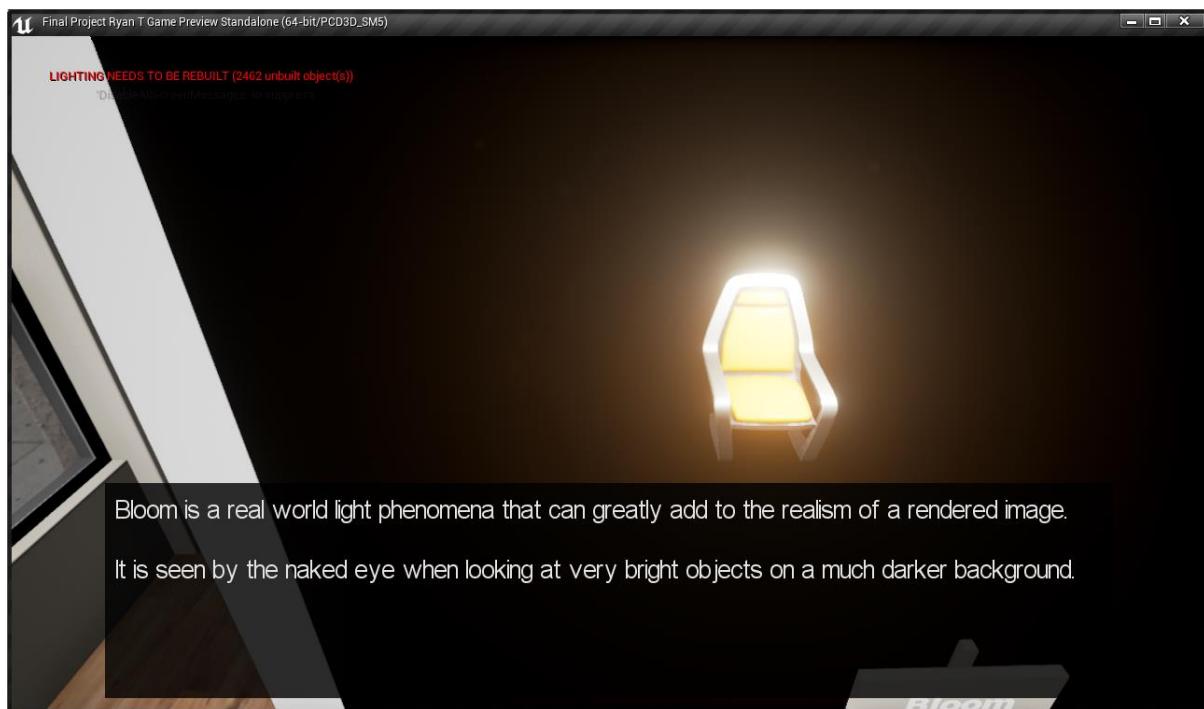


Figure 553

### Bloom Sequence: Cam moves into container around chair then back out again

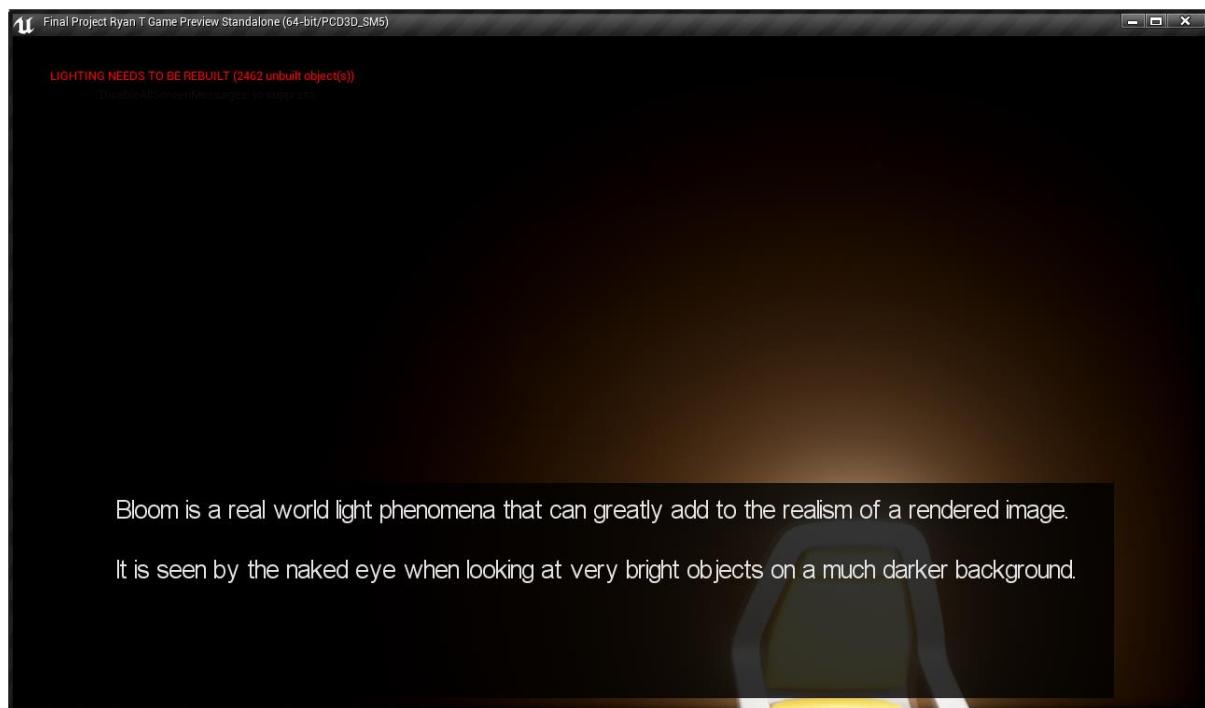


Figure 554

### Bloom Sequence: Cam moves into container around chair then back out again

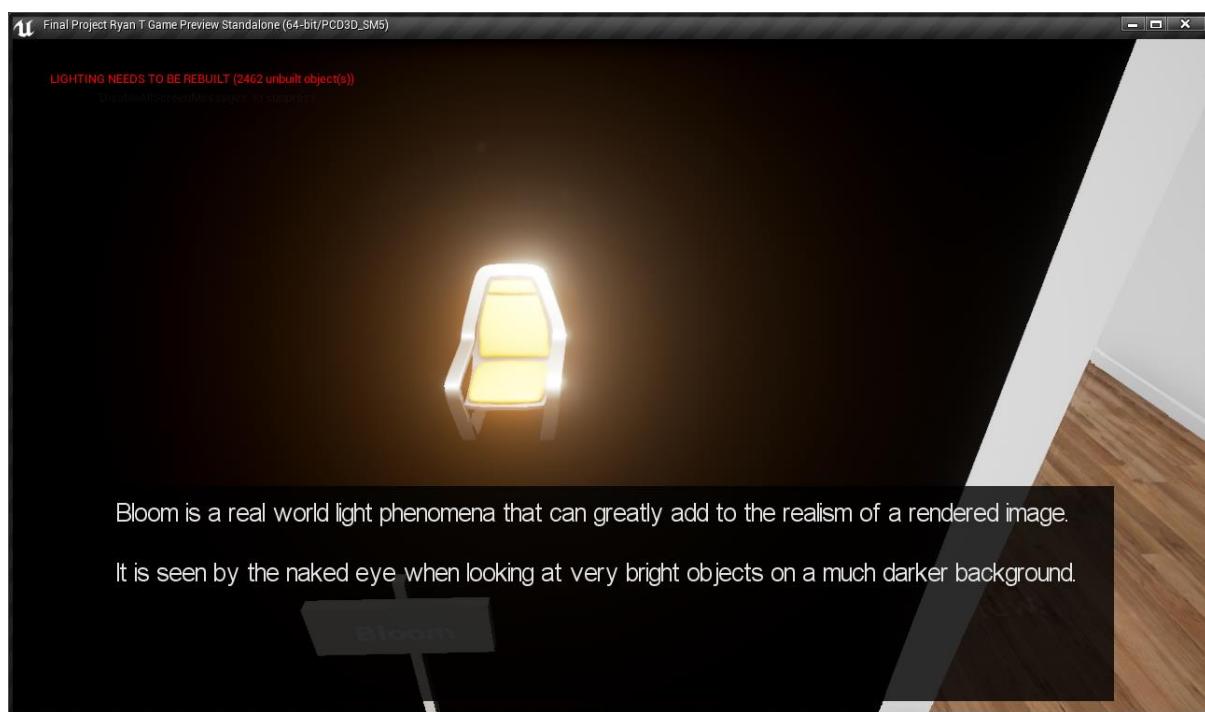


Figure 555

Bloom Sequence: Effect fades down, text goes off, demo ends

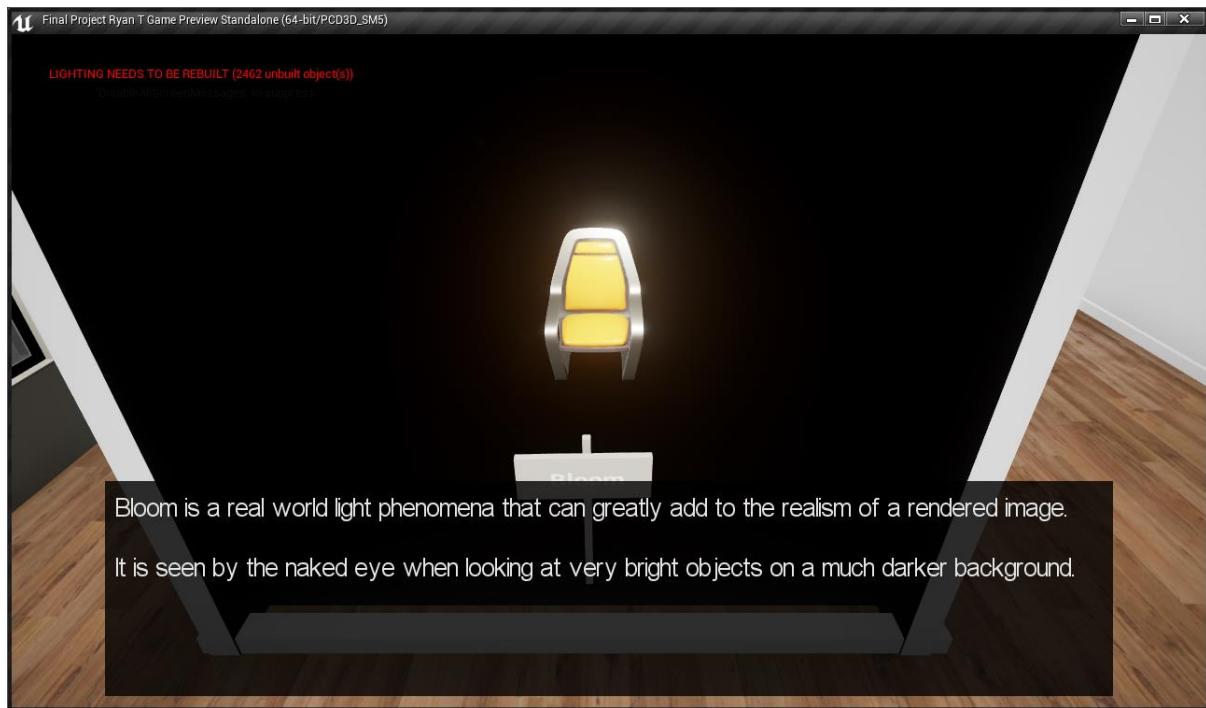


Figure 556

#### 6.8.6. Eye Adaption Demo

##### Sequenced Eye Adaption

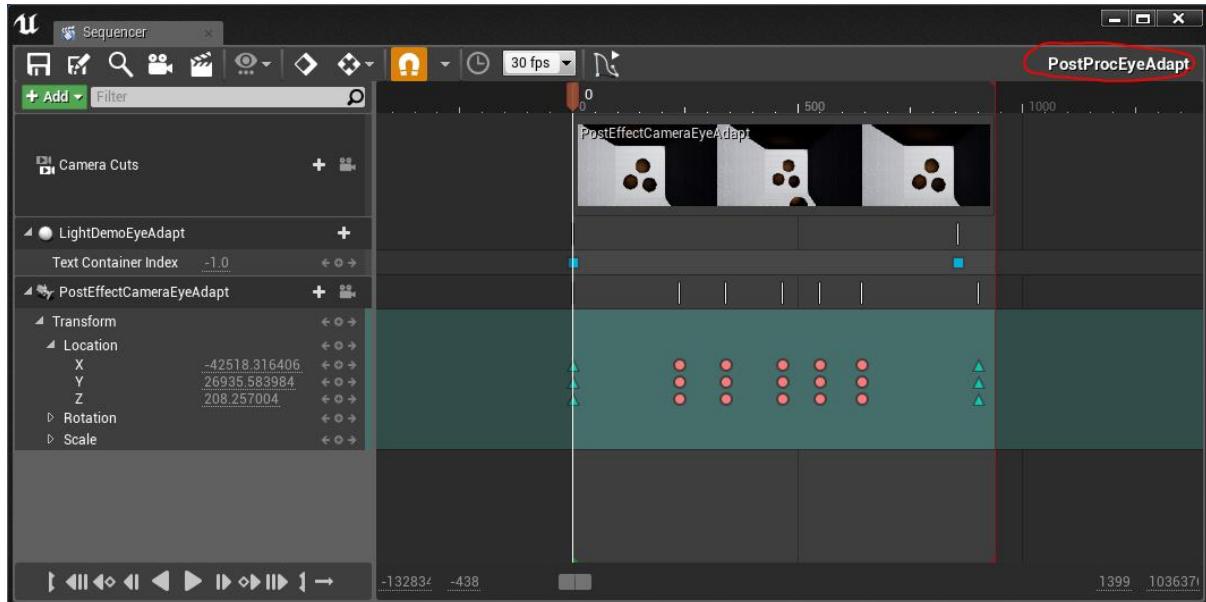


Figure 557

### Eye Adaption Sequence: Camera cuts, Text displayed

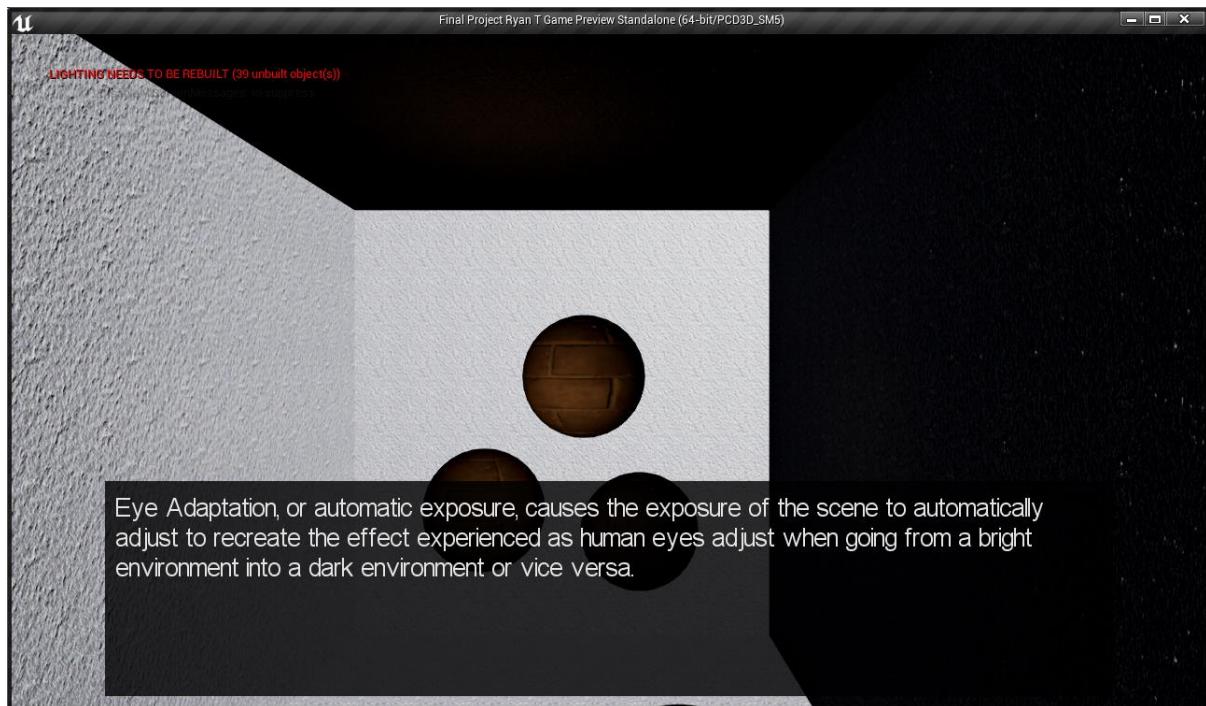


Figure 558

### Eye Adaption Sequence: Camera pans to the right showing the effect darken the scene as brighter background comes into view

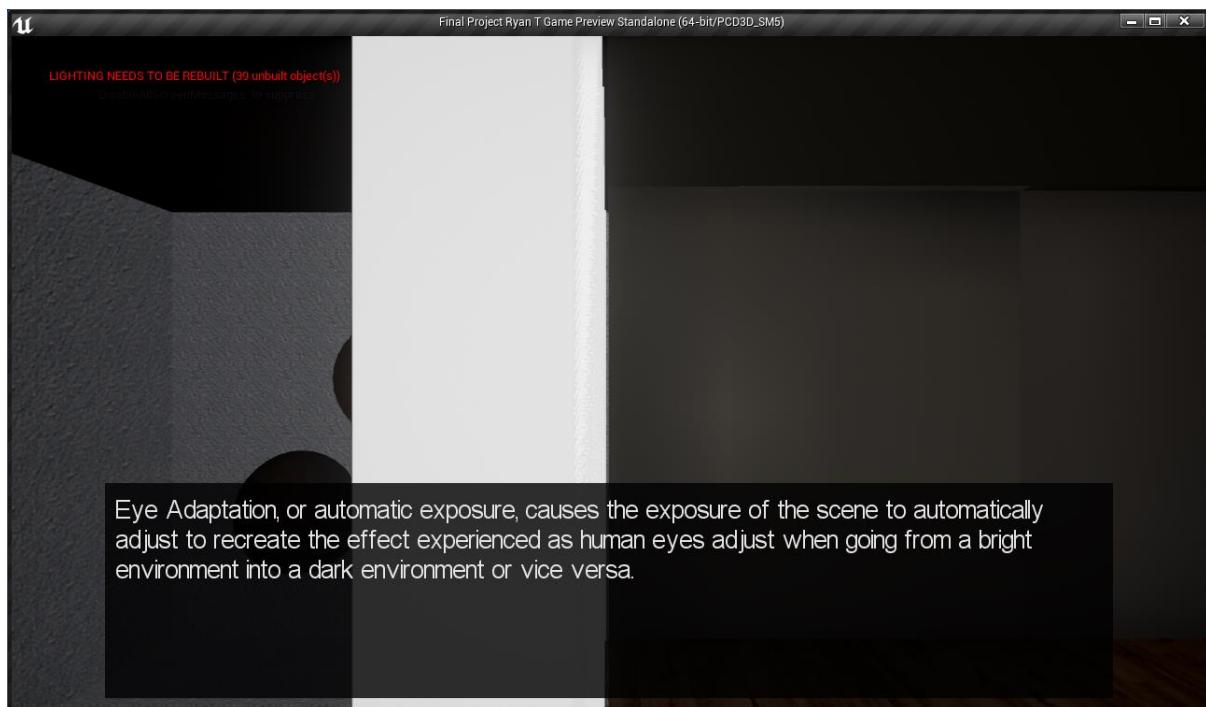


Figure 559

Eye Adaption Sequence: Camera pans to the back showing the effect darken the scene as brighter background comes into view

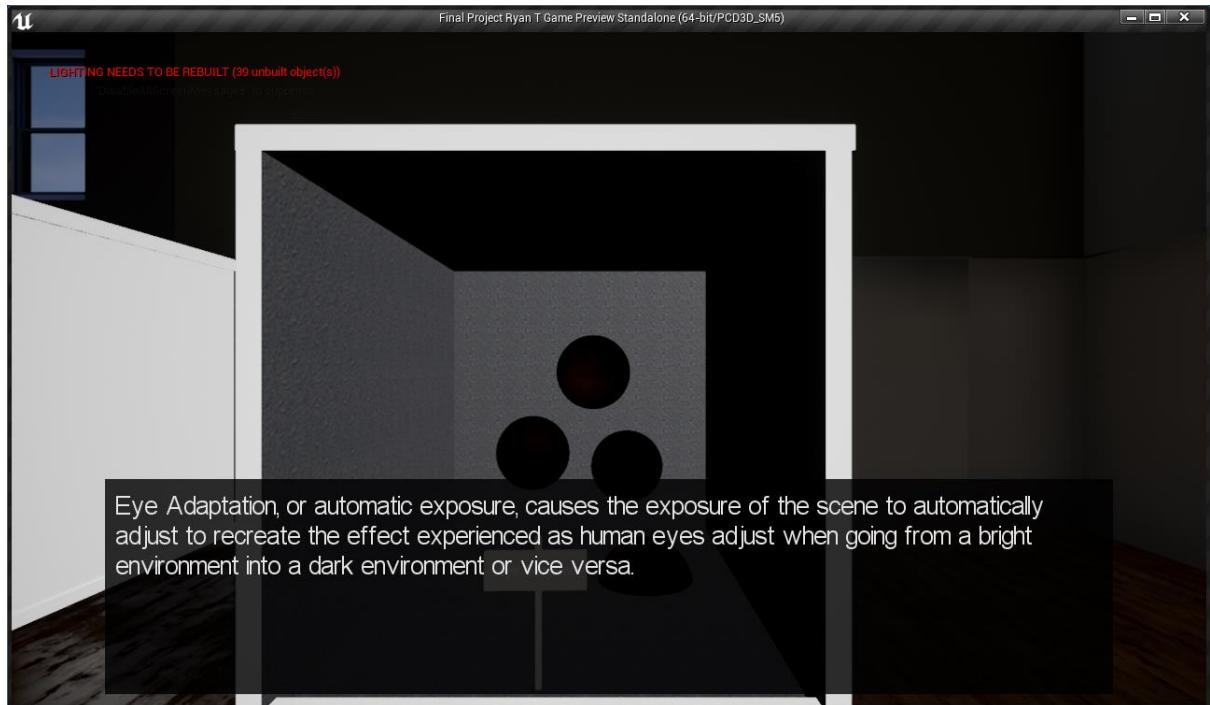
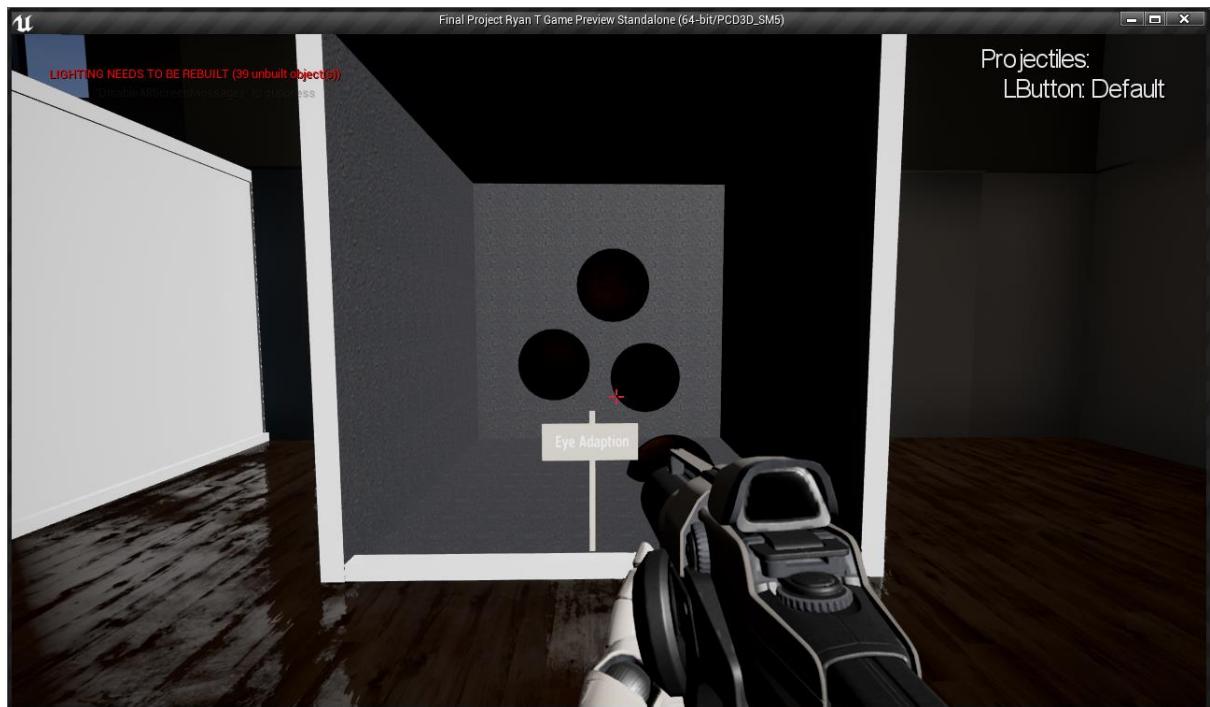


Figure 560

Eye Adaption Sequence: Sequence Ends



## 7. Testing

### 7.1. Application Testing

The application was tested using an agile approach throughout the implementation stage. Each feature in the app was worked on, and tested while being implemented. If any bugs were found, they would be fixed, and then development would move on. After a feature was completed, it was subject to a quick test again, if an error was found, it was fixed, and if it was fine, development moved on to the next stage.

The full testing log for the application can be found in appendix 10.4.

### 7.2. Peer Testing

To test how effective the developed application was in practice, and how comparable it is to forms of ‘analogue’ teaching, firstly a lecture was created. This lecture contained textual descriptions of all the things in the application along with a few diagrams. Essentially, this lecture was a written version of the developed application.

Two groups of 10 users each were then selected; the two groups consisted of friends and family members, the testing of the application was not to prove a research thesis, simply to test how the application performed as a learning tool, and how closely it compared to learning the same material from paper, this is after all, a software development project and not a research project.

The first group of people were given the application to use, they were given up to 1 hour to use the application; visiting the different areas, viewing the demonstrations etc. they were given the choice to finish early if they felt like they had a solid understanding of the content.

The second group of people were given a written lecture explaining all of the same principles as the application, but in written form. This group was also given an hour to learn the material.

Once both groups had finished, they were given a 10 minute break, then each person handed a questionnaire to fill out. A copy of this questionnaire can be found in appendix 10.3. Each question was worth a certain amount of marks, the whole questionnaire was worth 30 marks in total.

No personal information was collected from the participants, the questionnaires were completely anonymous, and none of the questionnaires were kept. The only data taken from this testing was a group of 20 numbers, a single number representing the total mark of each participant in the testing process (10 application, 10 lecture).

These results are shown in table 3 below. From these results you can see that participants who used the application had an average test score of ~21/30, while the participants who took the lecture only had an average score of ~20/30. It is also evident that participants using the application seemed to have overall higher scores than those who took the lecture; with a mode of 27 for those using the application, and a mode of only 20 for those taking the lecture. From these results, we can see that participants using the application appear to have a deeper understanding of the material than those who didn’t, even if it is only marginally so.

Application		Lecture	
1	27/30	1	20/30
2	14/30	2	17/30
3	23/30	3	20/30
4	11/30	4	25/30
5	28/30	5	10/30
6	27/30	6	20/30
7	19/30	7	18/30
8	26/30	8	28/30
9	21/30	9	30/30
10	13/30	10	11/30
TOTAL	209/300	TOTAL	199/300
MEAN	20.9	MEAN	19.9
MEDIAN	22	MEDIAN	20
MODE	27	MODE	20
STD. DEV	6.4	STD. DEV	6.5

*Table 3 - Peer Testing Results*

While the results of this testing appear promising, the sample size is far too small to gather any meaningful conclusions from. This test served only to demonstrate the effectiveness of the application on a small scale. Further, more comprehensive testing would be needed to determine if this application is truly “better” than traditional learning methods.

However, as a preliminary test, to give a general idea of the effectiveness of this application, this testing was a success; it demonstrated that the application is an effective learning tool that is on-par with learning methods such as lecturing.

## 8. Evaluations and Conclusions

As stated in other parts of the document, the application was tested using an agile approach, with each feature being constantly tested until it is complete, and when everything is complete the whole application being thoroughly tested for any errors. A variety of errors were found with the application throughout the development process, all of which are listed and explained in the testing log (Appendix 10.4). Only the more critical errors have been listed in here however, there were obviously hundreds more smaller errors, but these were not worth mentioning because of their irrelevancy.

Although the sample size for the peer testing of the application was very small, the testing demonstrated clearly that the application is efficient at demonstrating the techniques it was designed to. People who used the application were very engaged in the content offered by the application and seemed very interested whilst using it. IN addition, after using the application participants performed very well on a small questionnaire in which they answered questions asking them to describe various techniques demonstrated to them in the application, on average achieving scores of 21 out of a possible 30.

Section 4.1 of this document states that the purpose of this application is:

*“...to demonstrate various principles used in video games to simulate physics and lighting/effects. To allow the user to interact with objects in the environment to show what these principles are, how they are simulated and why they are needed.”*

From the peer testing, we can gather that the application does indeed demonstrate the principles it set out to effectively, and many of the demonstration areas have interactive portions; although the application could have benefited from more interactive features. However, the application still shows the user what the principles are, and describes to the user how they work. Overall, the application does meet the purpose/aims set out in the requirements and design documentation.

Section 4.9 of this document states that the specific requirements of this application are:

1. *The application should have a controllable character that the player can manipulate and use to travel around the level and interact with the demonstrations.*
2. *The application should have three distinct areas, physics, lighting/effects, and sandbox.*
3. *The physics area should contain four sub-areas containing demonstrations for the four researched physics principles.*
4. *The lighting/effects area should contain four sub-areas containing demonstrations for the four researched lighting/effects principles.*
5. *The sandbox area should contain a mixture of the elements from the previous two areas, presented in an informal manner, meant to entice people into using the application at fresher's fairs and other events.*
6. *Each demonstration should be kept separate from others, i.e. in their own rooms/buildings, but all relating demos i.e. all in the same category, should be segregated together.*
7. *At the very least, each demonstration should provide a description of the demonstrated principle and show player how it works. Ideally there should also be an interactive element accompanying the demonstration to further help the player grasp the concept.*
8. *The playable level in the application should be easily expandable for adding more demonstrations/areas in the future.*

The application meets all of these requirements except number 5, requirements 3 and 4 are met, but only partially, this will also be discussed below.

Requirement number 5 states that the ‘Sandbox Area’ should contain a mixture of elements from both areas of the application, presented in an informal manner. In the version of the application completed with this report however, the sandbox area is empty. This was due to a lack of time in the implementation phase, meaning the sandbox area could not be populated. However, as stated in the design section of this document, the sandbox area is optional in the finished product, yet it still remains an empty area in the application. This was because while it would have taken time that wasn’t available to add things into the area, it would also take a lot of time to remove the area, and reorganise the level around the new space. Due to this, the area was left there, so that it may be added to in a future release of the application.

Requirement number 3 and 4 state that the physics and lighting areas should both contain four areas, one for each researched principle. As with the sandbox area, in the version of the application completed with this report, not all areas were able to be finished. The lighting and effects area has only 2 of the 4 demonstration areas completed, again this was due to the fact that time was running out by the end of the project and the areas could not be completed. The lighting area was left until last, considering there was little to no code associated with it, and as such was not able to be completed in the time that was left. The physics area has only 3 of the 4 areas completed, however this was not due to time constraints. The molecular simulation demonstration was technically impossible to implement efficiently in the way that was planned. Code exists for this area, and can be found in the code listing at the end of this document, it worked, but was extremely resource intensive and caused the application to virtually stop when executed. An alternative solution was planned – using a feature of unreal capable of rendering particles on the GPU instead of the CPU, but considering the time left, this was not possible, so the area was dropped.

The application fully meets five of the requirements, and partially meets two of them, only one of the requirements was not met, and this was an optional feature. Overall, the application definitely meets the aims, purpose and requirements set out in chapters 4 and 5 of this document.

Although all of the areas did not make it into the final application, what content was there was extensive, and considering the relatively short timeframe of the implementation of the project, arguably somewhat impressive that it contains what it does. Further to this, some of the demonstrations are very satisfying to watch, and look very lifelike (unfortunately this cannot be seen without the application). The cloth especially looks very impressive, almost surprisingly life-like and is very satisfying to watch blowing in the wind. The flocking birds are also a personal favourite, they almost look like a real flock of birds, and it is very fun to break flocks apart with the repelling projectile.

While there are some notable strengths to the application, there are also some downsides. Most notably the lack of polish in the application and level itself. Due to time constraints there was no time to add needed polish to the level such as ambient sounds, fixing some of the quirks with the shadows on the building assets, adding miscellaneous objects around the world etc. The application would be very different with a significant level of polish. As previously explained, not all of the demonstration areas were implemented, while there is still a lot of content to this application, it would have been nice to get the additional demonstrations added. Furthermore, of the demonstrations that are implemented, not all of them have interactive portions, while the requirements stated that rooms only needed at least a demonstration to show the user a principle/technique, it would have been nice to have added a level of interactivity to every

demonstration area, the application would have been a lot more enjoyable with these additional layers of interactivity.

In order to fully determine the effectiveness of the application, and to conclude whether it is indeed better or worse than traditional lecture style learning, a much larger focus group would be needed. The focus group was intentionally kept relatively small for the peer testing of this application; the testing of the application was merely intended to verify that the application met the requirements set in chapters 4 and 5 of this document. Considering this application is not a safety critical system, and wouldn't contribute much to a valuable research domain in its current form, the amount of people in the focus group was kept as small as possible to still be able to come up with accurate results.

A Gantt chart was created in the early stages of this project outlining a proposed timeline for the duration of the project, this can be found in appendix 10.2. This Gantt chart was followed as closely as possible throughout this project, but in the end the project went on a very different path. The final version of the project specification was completed on time; this was the first stage of the project before anything else. The research was planned to last for four weeks, starting on October 30<sup>th</sup>, but due to a coursework that was due in soon after then, it was not started until November 9<sup>th</sup> – just under two weeks behind schedule. This also pushed back the implementation stage by two weeks. Implementation was started around December 11<sup>th</sup>, and the Unreal Project and level was laid out fairly quickly, however, and exam and coursework due after the Christmas break made progress relatively slow and not much was done in the programming side of the project. After the exam and coursework was over with, progress really took off, with almost two months of time to focus on development in between other set work. Lots of the project was completed in this time, but again progress slowed when other coursework was due. Development finished in early April – a week or so behind schedule, and the report compiled over the rest of April. While the project strayed from the planned route as shown by the Gantt chart, all lost time was made up for, and the work schedule adapted to finish on time.

## 9. References

- [1] - Attention span during lectures: 8 seconds, 10 minutes, or more? Neil A. Bradbury
- [2] - <https://unity3d.com/unity/features/multiplatform>
- [3] - <https://store.unity.com/products/unity-personal>
- [4] - <https://store.unity.com/products/unity-plus>
- [5] - <https://store.unity.com/products/unity-pro>
- [6] - <https://store.unity.com/products/unity-enterprise>
- [7] - <http://uk.ign.com/articles/2009/11/05/epic-games-announces-unreal-development-kit-powered-by-unreal-engine-3>
- [8] - <https://www.unrealengine.com/en-US/eula>
- [9] - <https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard>
- [10] - <https://docs.unrealengine.com/en-us/Engine/Content/ContentStandards>
- [11] - <https://docs.unrealengine.com/en-us/Engine/Performance/Guidelines>
- [12] - <http://www.songho.ca/math/plane/plane.html>
- [13] - Reynolds, Craig W. (1987). "Flocks, herds and schools: A distributed behavioral model.". ACM SIGGRAPH Computer Graphics. 21. pp. 25–34.

- Akenine-Möller, T., Haines, E. and Hoffman, N. (2016). Real-Time Rendering, Third Edition. 3rd ed. Baton Rouge: CRC Press.
- Birn, J. (2014). Digital lighting and rendering. 3rd ed. Berkeley, Calif: New Riders.
- Bourg, D. (2011). Physics for game developers. 1st ed. Sebastopol: O'Reilly.
- Bradbury, N. (2016). Attention span during lectures: 8 seconds, 10 minutes, or more?. Advances in Physiology Education, 40(4), pp.509-513.
- Bunce, D., Flens, E. and Neiles, K. (2010). How Long Can Students Pay Attention in Class? A Study of Student Attention Decline Using Clickers. Journal of Chemical Education, 87(12), pp.1438-1443.
- Furió, D., Juan, M., Seguí, I. and Vivó, R. (2014). Mobile learning vs. traditional classroom lessons: a comparative study. Journal of Computer Assisted Learning, 31(3), pp.189-201.
- Lester, P. and King, C. (2009). Analog vs. Digital Instruction and Learning: Teaching Within First and Second Life Environments. Journal of Computer-Mediated Communication, 14(3), pp.457-483.
- Manap, A., Sardan, N. and Rias, R. (2013). Interactive Learning Application in Microbiology: The Design, Development and Usability. Procedia - Social and Behavioral Sciences, 90, pp.31-40.
- Papastergiou, M. (2009). Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation. Computers & Education, 52(1), pp.1-12.
- Violante, M. and Vezzetti, E. (2013). Virtual interactive e-learning application: An evaluation of the student satisfaction. Computer Applications in Engineering Education, 23(1), pp.72-91.

- <http://www.songho.ca/math/plane/plane.html>
- <https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Overview>
- <https://docs.unrealengine.com/en-US/Engine/Physics/Collision/HowTo>
- <https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Reference>
- <https://viscomp.alexandra.dk/?p=147>
- <https://github.com/cyclejs-community/boids/blob/master/README.md>
- [http://www.cs.cmu.edu/afs/cs/academic/class/15462-s10/www/lectures/Lecture24\\_flocking.pdf](http://www.cs.cmu.edu/afs/cs/academic/class/15462-s10/www/lectures/Lecture24_flocking.pdf)
- <https://www.red3d.com/cwr/boids/>
- [https://www2.msm.ctw.utwente.nl/sluding/TEACHING/JMBC/2008/luding\\_2.pdf](https://www2.msm.ctw.utwente.nl/sluding/TEACHING/JMBC/2008/luding_2.pdf)
- <https://answers.unrealengine.com/questions/66842/powder-simulation-1.html>
- <https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/Lightmass/Basics>
- <https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightTypes>
- <https://docs.unrealengine.com/en-US/engine/rendering/lightingandshadows/lighttypes/directional>
- <https://docs.unrealengine.com/en-US/engine/rendering/lightingandshadows/lighttypes/point>
- <https://docs.unrealengine.com/en-US/engine/rendering/lightingandshadows/lighttypes/spot>
- <https://docs.unrealengine.com/en-US/engine/rendering/lightingandshadows/lighttypes/skylight>
- <https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/Basics>
- <https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/LightMobility>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/LightMobility/StaticLights>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/LightMobility/StationaryLights>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/LightMobility/DynamicLights>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/LightmassPortals>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/AntiAliasing>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/AutomaticExposure>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/DepthOfField>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/LensFlare>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/Vignette>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/PostProcessEffects/Bloom>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/Materials>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/PhysicallyBased>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/MaterialInputs>
- <https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/MaterialProperties>
- <https://docs.unrealengine.com/en-us/Resources>Showcases/Reflections>
- <https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/ReflectionEnvironment>

## 10. Appendices

### 10.1. Monthly Meeting Forms

#### 10.1.1. November



#### **6000PROJ Final Year Project Monthly Supervision Meeting Record**

Student: Ryan Tinman ..... Date: November

Main issues / Points of discussion / Progress made
<ul style="list-style-type: none"><li>• Researched various physics and lighting concepts to put in to tech demo</li><li>• Started designing the demo and how all the researched concepts will fit in.</li><li>• Installed the same version of Unreal Engine that is on university computers</li><li>• Became familiar with Unreal Engine in preparation for implementation</li></ul>
Actions for the next month
<ul style="list-style-type: none"><li>• Working start on the demo; First stages of prototype (Blocking out level, making character controller etc.)</li><li>• Start implementing the design; adding researched concepts etc.</li></ul>
Deliverables for next time
<ul style="list-style-type: none"><li>• I would like to see your designs the next time we meet. I would also like to see <u>some</u> Problem Analysis &amp; Domain Research at this stage.</li><li>• Demonstrate that you have looked at other packages apart from Unreal.</li><li>• Do think about Testing of the Software Product (i.e. keep a log).</li></ul>
Other comments
<ul style="list-style-type: none"><li>• You do appear to have started off your project in the right direction.</li></ul>
Supervisor signature: Mike Baskett .....
Student signature: Ryan Tinman.....



**6000PROJ Final Year Project**  
**Monthly Supervision Meeting Record**

Student: Ryan Tinman .....

Date: December

Main issues / Points of discussion / Progress made
<ul style="list-style-type: none"><li>Made first draft of design document</li><li>Researched games engines and made a start on the Problem Analysis &amp; Domain Research document</li><li>Completed tutorials on Unreal and C++, so software could be developed efficiently</li><li>Started working on the software in Unreal</li></ul>
Actions for the next month
<ul style="list-style-type: none"><li>Design document and Problem Analysis &amp; Domain Research should be nearing completion</li><li>Software should be in a playable state with a good number of things from the design document implemented.</li><li>A testing log will be kept throughout development to track progress</li></ul>
Deliverables for next time
<ul style="list-style-type: none"><li>Your design document needs to show what you are actually intend to achieve in Unreal – a walkthrough would be nice.</li><li>Start implementing in Unreal</li><li>Get <u>screenshots</u> of everything you do in Unreal.</li></ul>
Other comments
<ul style="list-style-type: none"><li>Consider who you are going to use to test your project on and how you are going to get ethical approval.</li></ul>
Supervisor signature: Mike Baskett.....
Student signature: Ryan Tinman .....



**6000PROJ Final Year Project  
Monthly Supervision Meeting Record**

Student: Ryan Tinman .....

Date: January

**Main issues / Points of discussion / Progress made**

- Unreal level progress: Landscape modelled and textured, level areas whiteboxed. Level is in a playable state, however not many demonstrations have been implemented.
- Designs for each area nearing completion.
- Screenshots and testing log have been kept, and updated throughout the implementation of the software.
- Level has a starting area, an area for the physics demonstrations, an area for the lighting demonstrations, and a sandbox area, which will contain a mixture of elements. Its purpose being to get people interested in viewing the software and allowing them to experiment with the different principles demonstrated in the software, before looking at them in more detail with explanations.

**Actions for the next month**

- Software should be nearing completion; a good number/most of the things from the design document implemented.
- Design document should also be nearing completion, with walkthrough discussed in last month's meeting included.
- Ethical approval

**Deliverables for next time**

- You need to start writing up your report.
- I need to see your design documentation and your screenshots.
- Please show me a demo of what you have implemented next month meeting in the lab.

**Other comments**

- We will need to discuss ethical approval in next month's meeting.
- Email me when you have uploaded the design document and screenshots onto Canvas before the next meeting.

Supervisor signature: Mike Baskett .....

Student signature: Ryan Tinman .....

#### 10.1.4. February



### 6000PROJ Final Year Project Monthly Supervision Meeting Record

Student: Ryan Tinman.....

Date: February

<b>Main issues / Points of discussion / Progress made</b>	
<ul style="list-style-type: none"><li>• Unreal level physics area almost complete</li><li>• Just have to fix some quirks with the code for the cloth/rope.</li><li>• Designs are ongoing, all areas that I have implemented are designed, I have ideas for the one or two other areas for the physics area, but they are not written down.</li><li>• Screenshots are being kept, and the testing log is being updated regularly.</li></ul>	
<b>Actions for the next month</b>	
<ul style="list-style-type: none"><li>• Software should be completed by the end of March.</li><li>• I will make a start on the report within the next week or so, filling it out in tandem with the software development. After the software is completed at the end of March, I will have most of April to finalise everything and make sure all reports are up to scratch.</li><li>• Other documents (Design, literature review etc.) will also be worked on, and nearing completion by the end of March.</li><li>• Ethical Approval?</li></ul>	
<b>Deliverables for next time</b>	
<b>Other comments</b>	
<ul style="list-style-type: none"><li>• Not sure if you need ethical approval.</li><li>• You need to start on the report – screenshots are needed.</li><li>• A working demo is needed for the presentation.</li></ul>	
Supervisor signature: Mike Baskett.....	
Student signature: Ryan Tinman .....	

10.1.5. March



## 6000PROJ Final Year Project Monthly Supervision Meeting Record

Student: Ryan Tinman.....

Date: March

<b>Main issues / Points of discussion / Progress made</b>
<ul style="list-style-type: none"><li>• Software on track to be completed by the end of March.</li><li>• Laid out the report with headings, title page etc.</li><li>• Screenshots and testing plan are continually being updated.</li><li>• Design documents and other documents are being worked on, will be nearing completion by April meeting.</li><li>• Need to discuss software testing and ethical approval.</li></ul>
<b>Actions for the next month</b>
<ul style="list-style-type: none"><li>• Software will be completed.</li><li>• All design work and research will be completed/in final draft.</li><li>• Final report will be nearing completion.</li><li>• The rest of April will be used to finalise everything, and to make sure everything is up to scratch, and on track to get the best grade possible out of what is done.</li></ul>
<b>Deliverables for next time</b>
<ul style="list-style-type: none"><li>• Your final report is no way near to completion. It needs a title, references, etc. It is not clear what you are actually trying to achieve.</li><li>• Screenshots needed for Unity and Unreal. Try to use first person only – Do not use 'I'</li><li>• Define HUD and other acronyms.</li></ul>
<b>Other comments</b>
Supervisor signature: Mike Baskett .....
Student signature: Ryan Tinman .....

## 10.2. Gantt Chart

### 10.3. Peer Testing Questionnaire

## Peer Testing Questionnaire

**Q:** Give a simple explanation of how collision is detected between a line and a triangle

---

---

**Q:** What are the four checks needed to detect this collision?

1. \_\_\_\_\_
2. \_\_\_\_\_
3. \_\_\_\_\_
4. \_\_\_\_\_

**Q:** Give a brief explanation of the stages of simulating cloth

---

---

**Q:** What are the four rules of flocking?

1. \_\_\_\_\_
2. \_\_\_\_\_
3. \_\_\_\_\_
4. \_\_\_\_\_

**Q:** Name three types of light.

1. \_\_\_\_\_
2. \_\_\_\_\_
3. \_\_\_\_\_

**Q:** What is Anti-Aliasing?

---

---

**Q:** What is Depth of Field?

---

---

**Q:** What is Vignette?

---

---

**Q:** What is Eye Adaption?

---

---

**Q:** What is Lens Flare?

---

---

**Q:** What is Bloom?

---

---

#### 10.4. Testing Log

No.	Feature	Problem Description	Problem Solution
1	General	The Created C++ actor class didn't display transform tool or widget in the editor.	<p>Added the following code to attach a USceneComponent to the actor.</p> <pre>RootComponent = CreateDefaultSubobject&lt;USceneComponent&gt;(TEXT("SceneComponent"));</pre> <p>This is a requirement in For Unreal engine to place an actor in the world. All actors that are placed in the world need a USceneComponent component, this contains that the transform (Position, Scale and Rotation)</p>
2	RopeSim	Rope Collision not working - game registering collision, but no force applied to rope.	<p>Code determining collision was using the location of the colliding object, rather than the position of the actual collision point. The Unreal Engine ray trace function returns its results in a structure. This contains things such as Collision impact point, Impact normal, Start/End point for ray trace etc. On positive collision, the ray trace end point was being used the not the impact point when deciding when the mesh should be moved back to.</p>
3	RopeSim	Collision for rope when being dragged across cube's corner is spherical rather than a cube.	<p>This error was not fixed</p>
4	Wind	When implementing wind force, tried to use Unreal's WindDirectionalSource object, GetWindParameters does not exist in version 14.4.3.	<p>Had to create custom wind class. This uses cross modulating sine waves with an offset based on the world position the wind is being sampled from. This gives a less uniform look to the wind.</p>
5	Rope	Rope was jumping up and down on Z axis, with variations in frame rate.	<p>The first vertex (The one attached to rope object) was simulating and moving away from attachment point. Changed the simulate function so that simulation on any attached vertex is disabled.</p>
6	ClothSim	Constraint not working correctly. Rather than top edge vertices of cloth locking, the left edge is.	<p>Rather than constraining next vertex to previous vertex (Constraining current vertex, to right vertex), add 50% of constraint to both vertices to pull both in to centre instead of pulling them to the left. (Found after some research into cloth simulation techniques, previous method worked for rope, but not for cloth, which is why it wasn't done initially)</p>

7	RopeSim & ClothSim	Collision with vertices not working; collision components not being created.	To create components at runtime in Unreal Engine you must use ConstructObject NOT NewObject. The fix was a simple replacement, changing the line of code where they are created (See below).  Old: <code>UVerletSphereComponent* Sphere = NewObject&lt;UVerletSphereComponent&gt;(GetWorld(), TEXT("CollisionComponent"));</code>  New: <code>UVerletSphereComponent* Sphere = ConstructObject&lt;UVerletSphereComponent&gt;(UVerletSphereComponent::StaticClass(), this, TEXT("CollisionComponent"));</code>
8	RopeSim & ClothSim	Force not being applied to vertices in cloth or rope.	This was because it was calling an Unreal Engine function called AddForce, not custom one which had the same name.  Added "ParentVert->" before calling custom AddForce. VerletSphere has its own AddForce (This is the one in Unreal used for its physics, but custom one is being used on the verts).  ParentVert points it to the vertex that the sphere component belongs to.  Old: VerletSphere->AddForce(Hit.ImpactNormal * 10000.0f);  New: VerletSphere->ParentVert->AddForce(Hit.ImpactNormal * 10000.0f);
9	ClothSim	Cloth attachments of vertices other than the top row seem to attach the incorrect vertices.	The calculation in the function UpdateLockedVerts that converted a linear vertex index into a vertical and horizontal index was wrong. Fixed miscalculation of horizontal and vertical index from a vertex index.
10	RopeSim	The rope appears to be flat from certain orientations.	There was a missing vector normalise in the BuildMesh and UpdateMesh functions. This was causing skewing of the x, y, z offset that was used to give the rope its thickness.
11	General	References to actors added to level sequences are lost when the sequence is saved, then reloaded.	Unreal levels can consist of many sub-levels to allow modular sections to be created and reused in other worlds. There is a bug in this version of Unreal which means that any actors added to a level sequence that are not in the persistent level (the root level) will lose their reference next time the sequence is reloaded. Moved actors used in all level sequenced into the persistent level.

12	Text Container Component	When adding TextContainerIndex from the TextContainerComponent class to a level sequence timeline, no keys can be added.	Unreal timelines only allow bool, vector or float type variables to be added. Had to change TextContainerIndex from an int to float type to allow keys to be added in level sequencer timelines.
13	Text Container Component	Any text added to the text render component classes text list which contained carriage returns produces double carriage returns.	The function being used to display the text requires just line feeds, not carriage return and line feed. Added code to filter CR characters but leave LF to prevent engine showing double line spacing.
14	Level Sequences	Random crashes after playing a level sequence in any area.	Added UPROPERTY to cloth SequencePlayer member of any class using sequences to enable UE4 garbage collection for that object. Pointer to objects using the UPROPERTY macro are reference counted by Unreal Engine and are garbage collected once the object is not referenced. Without this macro, the level sequence was freed up when it finished, but left a “stale” pointer in the class. This was then referenced by the code causing a crash.
15	HUD Text	Crash after displaying text in the HUD subtitle text.	Added check to make sure HUD subtitle text was not updated unless TextContainerIndex $\geq 0$ . A value of -1 is used to denote not text displayed. This was being used to index the text list with -1, causing the crash.
16	HUD Text	Text sometimes isn't displayed during level sequences.	Multiple actors were accessing the HUD subtitle text at the same time, one usually displaying text, the others clearing it. So the HUD text was being cleared out by other actors.  Added additional checks when updating the HUD text to only do so if TextContainerIndex $\geq 0$ and the actor is currently playing a level sequence. This means that only one actor is accessing the HUD text at once.
17	Level Sequencing	Demo level sequences are being triggered by firing projectiles into the trigger volume.	Added checks to stop level sequences being triggered by projectiles or other actors. Only Characters can trigger them now.

19	Collision Demo	Collision demo sequence fails to stop when exiting the trigger volume.	If SequencePlayer Stop function is called when the sequence is pause, it will fail to stop. The sequence must be un-pause before calling Stop. Fixed by continuing playing from the paused position before calling Stop.
----	----------------	--	--

## 10.5. Source Code Listing

### Collision Demo

*CollisionDemo.h*

```
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "GameFramework/Actor.h"
#include "Components/TextRenderComponent.h"
#include "Components/SphereComponent.h"
#include "LevelSequence.h"
#include "LevelSequenceActor.h"
#include "LevelSequencePlayer.h"
#include "RyanTCode/Misc/TextContainerComponent.h"
#include "CollisionDemo.generated.h"

UENUM()
details panels
enum class eCollisionRenderMode : uint8
{
    FacePlaneWithIntersect,
    FacePlaneAndEdgesWithIntersect,
};

UCLASS()
class FINALPROJECTRYANT_API ACollisionDemo : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ACollisionDemo();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

private:
    UPROPERTY(EditAnywhere, Category = "Collision Demo|Verts", Meta = (MakeEditWidget = true))
    TArray<FVector> LocalSpaceVerts;
```

```

// Line start point
UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
    FVector StartPoint;

// Line end point
UPROPERTY(EditAnywhere, Category = "Collision Demo|Points", Meta = (MakeEditWidget = true))
    FVector EndPoint;

// Offset for line to allow the level sequence to update it's position in the timeline
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Points")
    FVector WorldSpaceLineOffset = FVector(0.0f, 0.0f, 0.0f);

// Pointer for the trigger volume component
UPROPERTY(EditAnywhere, Category = "Collision Demo|Trigger")
    USphereComponent* TriggerSphere = nullptr;

// Pointer for the attached level sequence
UPROPERTY(EditAnywhere, Category = "Collision Demo|Trigger")
    ULevelSequence* LevelSequence = nullptr;

// Mode for how the triangle is drawn.
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
    eCollisionRenderMode CollisionRenderMode = eCollisionRenderMode::FacePlaneWithIntersect;

// Alpha value for the face planes opacity
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
    float FacePlaneAlpha = 0.04f;           // 10

// bool to turn on/off the intersection drawing
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
    bool bRenderIntersect = false;

// Index of current textcontainer text displayed in the HUD SubTitle area
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Text")
    float TextContainerIndex = -1;          // -1 default = No Text

// Timer to allow the playing level sequence the be pause for this many seconds
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Trigger")
    float LevelSequencePauseTime = 0.0f;

// The the timeline position where pause began to allow later un-pausing
float LevelSequencePauseStartTime = 0.0f;

```

```

// Bool to turn on/off triangle edge plane drawing
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
    bool bRenderAllEdgePlanes = false;

// Bool to turn of/off line drawing
UPROPERTY(EditAnywhere, Interp, Category = "Collision Demo|Rendering")
    bool bRenderLine = false;

// List for the world space verts
TArray<FVector> WorldSpaceVerts;

// Lisl of edge plane
TArray<FPlane> EdgePlanes;

// the plane for the triangles face
FPlane FacePlane;

// A pointer to a UTextRenderComponent that is use to show info such as the plane equation results etc
UPROPERTY() // UPROPERTY tells engine to consider for garbage
collection by adding a UPROPERTY
UTextRenderComponent* RightText = nullptr;

// A pointer to a UTextContainerComponent that is use to store all string to be shown in the HUD SubTitle area
UPROPERTY()
UTextContainerComponent* TextContainer = nullptr;

// Pointer to the sequence player. Used to control the sequence after play has began. Pause etc...
UPROPERTY()
ULevelSequencePlayer* SequencePlayer = nullptr;

// Callbacks for enter/exit volume

UFUNCTION() // UFUNCTION tells the engine that this function can be used as a callback (Required by
Unreal)
    void EnterSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);

UFUNCTION()
    void ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex);

private:
    void DrawTriangle();
    void UpdateWorldSpaceVerts();

```

```

void AddVert();
void ClearVerts();
void UpdatePlanes();
void DrawDebugPlanes();
void PerformCollisionCheck();
void OnEndSequnceEvent();
void ResetSequence();
};

CollisionDemo.cpp
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "DrawDebugHelpers.h"
#include "CollisionDemo.h"
#include "FinalProjectRyanTHUD.h"

// *****
// Constructor
// *****
ACollisionDemo::ACollisionDemo()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // USphereComponent is used to create a trigger volume around the actor (collision demo).
    TriggerSphere = CreateDefaultSubobject<USphereComponent>(TEXT("SphereComponent"));
    RootComponent = TriggerSphere;
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void ACollisionDemo::BeginPlay()
{
    Super::BeginPlay();

    // Register the enter and exit callbacks for the sphere volume
    TriggerSphere->OnComponentBeginOverlap.AddDynamic(this, &ACollisionDemo::EnterSphereVolume);
    TriggerSphere->OnComponentEndOverlap.AddDynamic(this, &ACollisionDemo::ExitSphereVolume);

    // Transforms local space verts into worldspace verts
    UpdateWorldSpaceVerts();
}

```

```

// Get a list of all components
TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextRenderComponent::StaticClass());

// Scan the list looking for text components
for (int i = 0; i < TextComponents.Num(); i++)
{
    // Fill in pointer to the right text component
    if (TextComponents[i]->GetName() == "RightText")
        RightText = Cast<UTextRenderComponent>(TextComponents[i]);

    // Fill in pointer to the text container component (This is used for descriptions)
    if (TextComponents[i]->GetName() == "TextContainer")
        TextContainer = Cast<UTextContainerComponent>(TextComponents[i]);
}

// Reset everything to do with the sequence
ResetSequence();
}

// *****
// Tick - Called every frame
// *****
void ACollisionDemo::Tick( float DeltaTime )
{
    Super::Tick( DeltaTime );

    UpdateWorldSpaceVerts();
    UpdatePlanes();
    PerformCollisionCheck();
    DrawTriangle();

    if ( SequencePlayer )                                // Are we playing the level sequence?
    {
        if ( LevelSequencePauseTime )      // Sequence is paused?
        {
            // If not paused yet
            if ( !LevelSequencePauseStartTime )
            {
                // Pause it and record the time it was paused
                SequencePlayer->Pause();
                LevelSequencePauseStartTime = SequencePlayer->GetPlaybackPosition();
            }
        }
    }
}

```

```

        // Count down the timer
        LevelSequencePauseTime -= DeltaTime;

        if ( LevelSequencePauseTime <= 0.0f )
        {
            // When timer finished, reset everything and continue playing from old position
            LevelSequencePauseTime = 0.0f;

            SequencePlayer->SetPlaybackPosition(LevelSequencePauseStartTime + 0.1f);           // +0.1 to advance on
from pause key
            SequencePlayer->StartPlayingNextTick();
            LevelSequencePauseStartTime = 0.0f;
        }
    }
    else
    {
        // Sequence is not paused so Poll for sequence ending
        if ( !SequencePlayer->IsPlaying() )
            OnEndSequnceEvent();                                // Clean up sequence player
    }
}
else
{
    TextContainerIndex = -1;          // Turn off text box if there is no sequence playing
}
}

// *****
// EnterSphereVolume - Called when component enters collision volume
// *****
void ACollisionDemo::EnterSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

//    DrawDebugSphere(GetWorld(), GetActorLocation(), 20.0f, 8.0f, FColor::Orange, true, 5.0f);

    if (LevelSequence)
    {
        FLevelSequencePlaybackSettings Settings;
        Settings.LoopCount = 0;                                // No looping
        SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
    }
}

```

```

        SequencePlayer->Play();
    }

// *****
// ExitSphereVolume - Called when component exits collision volume
// *****
void ACollisionDemo::ExitSphereVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Bug Fix: Continue playing from old position before stopping otherwise Stop will fail...
    if ( LevelSequencePauseTime )      // Sequence is paused?
    {
        SequencePlayer->SetPlaybackPosition(LevelSequencePauseStartTime + 0.1f);           // +0.1 to advance on from pause
key
        SequencePlayer->StartPlayingNextTick();
    }

    if ( SequencePlayer )
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }

    // Reset everything to do with the sequence
    ResetSequence();

    // Reset the SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();
}

// *****
// OnEndSequnceEvent - Triggered when level sequence has finished playing
// *****
void ACollisionDemo::OnEndSequnceEvent()
{
    // Stop the sequence playing
    SequencePlayer->Stop();
    SequencePlayer = nullptr;
}

```

```

        // Reset everything to do with the sequence
        ResetSequence();
    }

// *****
// DrawTriangle - Draws the triangle
// *****
void ACollisionDemo::DrawTriangle()
{
    // Draw the triangle
    TArray<int> Indices;
    Indices.Add(0);
    Indices.Add(1);
    Indices.Add(2);
    DrawDebugMesh(GetWorld(), WorldSpaceVerts, Indices, FColor(255, 0, 25, 20), false, 0.0f, 0);

    // Draw the triangle outline.
    for (int i = 0; i < LocalSpaceVerts.Num(); i++)
    {
        int j = (i + 1) % LocalSpaceVerts.Num();                                // Gets the index of the next vert, and wraps round
back to the start                                                               // if the array goes out of
bounds.                                                                       
//        DrawDebugLine(GetWorld(), WorldSpaceVerts[i], WorldSpaceVerts[j], FColor::Red, false, 0.0f, 0, 2.0f);
//        DrawDebugLine(GetWorld(), WorldSpaceVerts[i], WorldSpaceVerts[j], FColor::Red, false, 0.0f, 0, 1.0f);
    }
}

// *****
// UpdateWorldSpaceVerts - Transforms local space verts into worldspace verts
// *****
void ACollisionDemo::UpdateWorldSpaceVerts()
{
    // If the number of local an world space verts don't match, then re-build the list
    if (WorldSpaceVerts.Num() != LocalSpaceVerts.Num())
    {
        ClearVerts();

        for (int i = 0; i < LocalSpaceVerts.Num(); i++)
            AddVert();
    }

    FTransform Transform = GetActorTransform();                                     // Get the transform for this ACollisionDemo actor
}

```

```

        for (int i = 0; i < LocalSpaceVerts.Num(); i++)
    {
        FVector Pos = Transform.TransformPosition(LocalSpaceVerts[i]);           // Transform each vert into World Space as
verts are in actor's local space.
        WorldSpaceVerts[i] = Pos;
    }
}

// *****
// AddVert - Adds verts
// *****
void ACollisionDemo::AddVert()
{
    WorldSpaceVerts.Add(FVector::ZeroVector);           // Added an empty vector (0,0,0) to WorldSpaceVerts,
                                                       // to increase it's size to match
LocalSpaceVerts

    EdgePlanes.Add(FPlane(0, 0, 0, 0));                // For each vert, add an edge plane
}

// *****
// ClearVerts - Clears all verts lists
// *****
void ACollisionDemo::ClearVerts()
{
    WorldSpaceVerts.Empty();                         // Empty WorldSpaceVerts list

    EdgePlanes.Empty();                            // Empty EdgePlanes list
}

// *****
// UpdatePlanes - Creates face and edge planes
// *****
void ACollisionDemo::UpdatePlanes()
{
    if (WorldSpaceVerts.Num() < 3)          // If there are less than three verts, return
        return;

    // Create face plane
    FacePlane = FPlane(WorldSpaceVerts[0], WorldSpaceVerts[1], WorldSpaceVerts[2]);

    // Create edge planes
    for (int i = 0; i < WorldSpaceVerts.Num(); i++)
    {
}

```

```

        int j = (i + 1) % WorldSpaceVerts.Num();                                // Gets the index of the next vert, and wraps round
back to the start                                                               // if the array goes out of
bounds.
        FVector EdgeDir = WorldSpaceVerts[j] - WorldSpaceVerts[i];
        EdgeDir.Normalize();

        FVector EdgeNormal = FVector::CrossProduct(FacePlane, EdgeDir);

        EdgePlanes[i] = FPlane(WorldSpaceVerts[i], EdgeNormal);
    }
}

// *****
// DrawDebugPlanes - Draws debug planes
// *****
void ACollisionDemo::DrawDebugPlanes()
{
    if (WorldSpaceVerts.Num() < 3)           // If there are less than three verts, return
        return;

    // Draw the face plane
    FVector Pos = (WorldSpaceVerts[0] + WorldSpaceVerts[1] + WorldSpaceVerts[2]) / 3.0f;
    FVector2D Ext(200, 200);
    DrawDebugSolidPlane(GetWorld(), FacePlane, Pos, Ext, FColor(66, 217, 255, 255.0f*FacePlaneAlpha) );

    if ( CollisionRenderMode == eCollisionRenderMode::FacePlaneAndEdgesWithIntersect )
    {
        // Draw the edge planes
        for (int i = 0; i < WorldSpaceVerts.Num(); i++)
        {
            int j = (i + 1) % WorldSpaceVerts.Num();                                // Gets the index of the next vert, and wrap
round back to the start                                                               // if the array goes
out of bounds.
            Pos = (WorldSpaceVerts[i] + WorldSpaceVerts[j]) / 2.0f;
            Ext = FVector2D(10, 100);

            DrawDebugSolidPlane(GetWorld(), EdgePlanes[i], Pos, Ext, FColor::Green);
        }
    }
}

```

```

// *****
// PerformCollisionCheck - Checks for line to triangle collision
// *****
void ACollisionDemo::PerformCollisionCheck()
{
    FTransform Transform = GetActorTransform();

    FVector WorldSpaceStartPoint = Transform.TransformPosition(StartPoint);           // Transforms StartPoint into World Space
    FVector WorldSpaceEndPoint = Transform.TransformPosition(EndPoint);               // Transforms EndPoint into World
Space

    WorldSpaceStartPoint += WorldSpaceLineOffset;                                     // Add offset to start and end points to allow them to
be animated from level sequencer
    WorldSpaceEndPoint += WorldSpaceLineOffset;

    if ( bRenderAllEdgePlanes )
    {
        // Draws stuff for the initial part of demo, before collision part begins...

        // Draw the face plane
        FVector Pos = (WorldSpaceVerts[0] + WorldSpaceVerts[1] + WorldSpaceVerts[2]) / 3.0f;
        FVector2D Ext(200, 200);
        DrawDebugSolidPlane(GetWorld(), FacePlane, Pos, Ext, FColor(66, 217, 255, 255.0f*FacePlaneAlpha) );

        // Draw the 3 edge planes
        for (int i = 0; i < EdgePlanes.Num(); i++)
        {
            int j = (i + 1) % WorldSpaceVerts.Num();                                // Gets the index of the next vert, and wraps
round back to the start                                                       // if the array goes
out of bounds.
            Pos = (WorldSpaceVerts[i] + WorldSpaceVerts[j]) / 2.0f;
            Ext = FVector2D(10, 100);

            DrawDebugSolidPlane(GetWorld(), EdgePlanes[i], Pos, Ext, FColor::Green);
        }

        // Render the line the we will be testing collision with
        if ( bRenderLine )
        {
            DrawDebugLine(GetWorld(), WorldSpaceStartPoint, WorldSpaceEndPoint, FColor::White, false, -1.0f, 0.0f, 2.0f);

            // Show both points of the line as text in world
            DrawDebugString(GetWorld(), WorldSpaceStartPoint, "P1" , nullptr, FColor::White, 0.0f);

```

```

        DrawDebugString(GetWorld(), WorldSpaceEndPoint, "P2" , nullptr, FColor::White, 0.0f);
    }
    return;
}

// Later part of demo were collision is being performed...

// Draw the face plane
FVector Pos = (WorldSpaceVerts[0] + WorldSpaceVerts[1] + WorldSpaceVerts[2]) / 3.0f;
FVector2D Ext(200, 200);
DrawDebugSolidPlane(GetWorld(), FacePlane, Pos, Ext, FColor(66, 217, 255, 255.0f*FacePlaneAlpha) );

// Get the signed perpendicular distance to FacePlane (- behind plane, + ahead of plane)
float DistFront = FacePlane.PlaneDot(WorldSpaceStartPoint);

// Get the signed perpendicular distance to EndPlane (- behind plane, + ahead of plane)
float DistBehind = FacePlane.PlaneDot(WorldSpaceEndPoint);

// Render the line we will be testing collision with
if ( bRenderLine )
{
    DrawDebugLine(GetWorld(), WorldSpaceStartPoint, WorldSpaceEndPoint, FColor::White, false, -1.0f, 0.0f, 2.0f);

    FString s;

    s = "P1 (";
    s.AppendInt( int(DistFront) );
    s.Append( ")" );
    DrawDebugString(GetWorld(), WorldSpaceStartPoint+FVector(0,0,-8), s, nullptr, FColor::White, 0.0f, true);

    s = "P2 (";
    s.AppendInt( int(DistBehind) );
    s.Append( ")" );
    DrawDebugString(GetWorld(), WorldSpaceEndPoint+FVector(0,0,-8), s, nullptr, FColor::White, 0.0f, true);
}

int Count = 0;                                // Count for the number of successful edge plane checks

if (DistFront >= 0.0f && DistBehind <= 0.0f)                                         // If the front point is
infront of plane, and rear point is behind plane
{
    FVector LineVector = WorldSpaceEndPoint - WorldSpaceStartPoint;
}

```

```

        float ProjDist = FVector::DotProduct(-LineVector, FacePlane);                                // Calculate intersections point of
line segment with face plane

        float Ratio = DistFront / ProjDist;

        FVector Intersection = WorldSpaceStartPoint + (LineVector * Ratio);

        for (int i = 0; i < EdgePlanes.Num(); i++)
        {
            int j = (i + 1) % WorldSpaceVerts.Num();                                              // Gets the index of
the next vert, and wraps round back to the start

            // if the array goes out of bounds.

            float DistToEdge = EdgePlanes[i].PlaneDot(Intersection);                            // Gets the signed perpendicular
distance to current edge plane (- behind plane, + ahead of plane)

            FColor Colour = FColor::Red;

            if (DistToEdge >= 0.0f)
            {
                Colour = FColor::Green;
                Count++;
            }
Count the number of edge plane tests that are positive
        }

        if (CollisionRenderMode == eCollisionRenderMode::FacePlaneAndEdgesWithIntersect)
        {
            Pos = (WorldSpaceVerts[i] + WorldSpaceVerts[j]) / 2.0f;
            Ext = FVector2D(10, 100);

            DrawDebugSolidPlane(GetWorld(), EdgePlanes[i], Pos, Ext, Colour);
        }
    }

    if (bRenderIntersect)
        DrawDebugSphere(GetWorld(), Intersection, 5.0f, 8.0f, Count == EdgePlanes.Num() ? FColor::Green : FColor::Red
);
}

// Fill in the text for RightTextContent
if (RightText)
{
    FString RightTextContent = "";

```

```

if ( CollisionRenderMode == eCollisionRenderMode::FacePlaneAndEdgesWithIntersect )
{
    RightTextContent = "Intersection point is within\n";
    RightTextContent.AppendInt(Count);
    RightTextContent.Append( " Edge Planes" );

    if (Count == EdgePlanes.Num())
        // If the point is within all edge
planes (Line intersected with triangle)
        RightTextContent.Append( "\nSuccess!" );
}
else
{
    RightTextContent = "P1 Plane Equ ax+by+cz+d = ";
    RightTextContent.AppendInt(int(DistFront));

    RightTextContent.Append("\nP2 Plane Equ ax+by+cz+d = ");
    RightTextContent.AppendInt(int(DistBehind));
}

RightText->SetText(FText::FromString(RightTextContent));
}

// Update the text container component
if (TextContainer && TextContainerIndex >= 0)
{
// TextContainer->SetTextFromList(TextContainerIndex);           // No longer use Text container render
TextContainer->SetTextFromList(-1);                                // Turn off 3d text

FString Temp = TextContainer->GetTextFromList(TextContainerIndex);

// (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
while ( true )
{
    int32 Index;
    if ( !Temp.FindChar( TCHAR('\r'), Index ) )
        break;
    Temp.RemoveAt( Index, 1 );
}

// Now using sub-title box in HUD
AFinalProjectRyanTHUD::SetString( Temp );
}
}

```

```

// *****
// ResetSequence - Reset everything to do with the sequence
// *****
void ACollisionDemo::ResetSequence()
{
    // Reset some stuff
    SequencePlayer = nullptr;
    LevelSequencePauseTime = 0.0f;
    LevelSequencePauseStartTime = 0.0f;

    // Reset the SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();
    TextContainer->SetTextFromList(-1);                                // Turn of text

    // Resset the line offset
    WorldSpaceLineOffset = FVector( 0.0f, 0.0f, 0.0f );

    // Reset the collision render mode
    CollisionRenderMode = eCollisionRenderMode::FacePlaneWithIntersect;

    // Reset flags for rendering some components
    bRenderIntersect = false;
    bRenderAllEdgePlanes = false;
    bRenderLine = false;

    // Reset the Right text components string to empty
    if (RightText)
        RightText->SetText(FText::FromString(""));
}

```

## Flocking

*FlockingAI.h*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

```

#include "GameFramework/Actor.h"
#include "Engine/TargetPoint.h"
#include "RyanTCode/Misc/TextContainerComponent.h"
#include "LevelSequence.h"

```

```

#include "LevelSequenceActor.h"
#include "FlockingAI.generated.h"

// Structure that defines a single bird

USTRUCT()
struct FBird
{
    GENERATED_BODY()

public:

    // Destructor
    ~FBird()
    {
        if (BirdMesh)
            BirdMesh->UnregisterComponent();
        BirdMesh = nullptr;
    }

    FVector Position;                                // Birds position
    FVector Direction;                             // Birds direction
    FVector Waypoint;                            // Current waypoint it is trying to head for
    float Speed;                                 // It's speed. Multiplied be Direction unit vector the
get it's velocity

    // Pointer to the birds 3d mesh
    UPROPERTY()
    UStaticMeshComponent* BirdMesh = nullptr;

    // Updates the birds 3d mesh transform from its position and direction
    void UpdateTransform()
    {
        // Make sure we have a mesh instance before updating transform
        if (!BirdMesh)
            return;

        FTransform Trans;

        // set the position in the transform
        Trans.SetLocation(Position);

        // Make the model face in the birds direction
    }
};

```

```

        FRotator Rotator = Direction.Rotation();
        Trans.SetRotation(FQuat(Rotator));

        // Apply the transform to the mesh
        BirdMesh->SetWorldTransform(Trans);
    }

};

UCLASS()
class FINALPROJECTTRYANT_API AFlockingAI : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFlockingAI();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called when the game ends or when destroyed
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason);

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    // The list of all birds in the simulation
    TArray<FBird> BirdList;

    // The closest the birds should get
    UPROPERTY(EditAnywhere, Category = "Flocking|Distances")
        float SpacingDistance = 100.0f;

    // The distance that tells a bird who are it's neighbours
    UPROPERTY(EditAnywhere, Category = "Flocking|Distances")
        float GroupingDistance = 200.0f;

    // Distance to be to a projectile before trying to avoid it
    UPROPERTY(EditAnywhere, Category = "Flocking|Distances")
        float AvoidProjectileDistance = 1200.0f;

    // list of all waypoints
    UPROPERTY(EditAnywhere, Category = "Flocking")
    TArray<ATargetPoint*> Waypoints;

```

```

// The resource used to create all bird meshes
UPROPERTY(EditAnywhere, Category = "Flocking")
    UStaticMesh* BirdMeshResource = nullptr;

// Container for all text in the demo
UPROPERTY()
    UTTextContainerComponent* TextContainer = nullptr;

// Index of current piece of text
UPROPERTY(EditAnywhere, Interp, Category = "Flocking")
    float TextContainerIndex = -1; // -1 default = No Text

// Pointer to the attached level sequence
UPROPERTY(EditAnywhere, Category = "Flocking")
    ULevelSequence* LevelSequence = nullptr;

// Pointer to the sequence player. Used to control the sequence after play has began. Pause etc...
UPROPERTY()
    ULevelSequencePlayer* SequencePlayer = nullptr;

// Bool to turn on/off grouping rule
UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
    bool bEnableGrouping = true;

// Bool to turn on/off spacing rule
UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
    bool bEnableSpacing = true;

// Bool to turn on/off alignment rule
UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
    bool bEnableAlignment = true;

// Bool to turn on/off drawing of birds direction and desired direction
UPROPERTY(EditAnywhere, Interp, Category = "Flocking|Flags")
    bool bShowBirdsHeadings = false;

private:
    UPrimitiveComponent* LastAddedComponent = nullptr;

// Pointer to the trigger volume component. Demo begins when it is entered
UPROPERTY(EditAnywhere, Category = "Flocking")
    UBoxComponent* TriggerVolume = nullptr;

```

```

private:
    void MoveBirds(float DeltaTime);

    void ShowWaypoints();

    FVector PickNewWaypoint(FVector OldWaypoint);

    TArray<FBird*> GetNeighbours(FBird& Bird);

    FVector GetDirectionToGroup(FBird& CurrentBird, TArray<FBird*>& Neighbours);

    FVector GetDirectionAwayFromNeighbour(FBird& CurrentBird, TArray<FBird*>& Neighbours);
    FVector GetAlignmentDirection(FBird& CurrentBird, TArray<FBird*>& Neighbours);

    FVector GetProjectileAvoidDirection(FBird& CurrentBird);
    void UpdateSubTitleText();
    void CreateBirds();
    void DestroyBirds();

    void OnEndSequnceEvent();

    UFUNCTION() // UFUNCTION tells the engine that this function can be used as a callback
    (Required by Unreal)
        void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
    OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
    UFUNCTION()
        void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
    OtherBodyIndex);
};


```

### *FlockingAI.cpp*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "DrawDebugHelpers.h"
#include "FlockingAI.h"
#include "FinalProjectRyanTProjectile.h"
#include "FinalProjectRyanTHUD.h"

DECLARE_CYCLE_STAT(TEXT("Flocking Tick"), STAT_FlockingTick, STATGROUP_Ryan); // Stat within group 'Ryan' See
"FinalProjectRyanT.h"

```

```

// *****
// RULES
// *****
// 1. Grouping
// 2. Alignment
// 3. Spacing

// *****
// Constructor
// *****
AFlockingAI::AFlockingAI()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));

    // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void AFlockingAI::BeginPlay()
{
    Super::BeginPlay();

    // Get a list of all UTTextContainerComponents
    TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTTextContainerComponent::StaticClass());

    if (TextComponents.Num() > 0)
        TextContainer = Cast<UTTextContainerComponent>(TextComponents[0]);           // Use the first one in list (should only
be one)

    TextContainerIndex = -1;

    // Setup the trigger volume
    TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);

    TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AFlockingAI::EnterVolume);
    TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AFlockingAI::ExitVolume);
}

```

```

}

// *****
// EndPlay - Called when the game ends or when destroyed
// *****
void AFlockingAI::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    DestroyBirds();           // Kill all active birds
}

// *****
// Tick - Called every frame
// *****
void AFlockingAI::Tick( float DeltaTime )
{
    SCOPE_CYCLE_COUNTER(STAT_FlockingTick);

    Super::Tick( DeltaTime );

    MoveBirds(DeltaTime);

    // ShowWaypoints();

    UpdateSubTitleText();

    // Poll sequence player in it is playing to see if it has completed.
    if (SequencePlayer && !SequencePlayer->IsPlaying())
        OnEndSequnceEvent();
}

// *****
// MoveBirds - Moves all birds in BirdList
// *****
void AFlockingAI::MoveBirds(float DeltaTime)
{
    for (int i = 0; i < BirdList.Num(); i++)
    {
        FBird& CurrentBird = BirdList[i];

        // Get a list of neighbouring birds
        TArray<FBird*> Neighbours          = GetNeighbours(CurrentBird);

        // Accumulate any direction changes needed
        FVector DirectionToGroup           = FVector(0,0,0);           // Initially zero vectors
}

```

```

FVector DirectionAwayFromNeighbour      = FVector(0,0,0);
FVector AlignmentDirection             = FVector(0,0,0);

// Grouping
if ( bEnableGrouping )
    DirectionToGroup = GetDirectionToGroup(CurrentBird, Neighbours);

// Spacing
if ( bEnableSpacing )
    DirectionAwayFromNeighbour = GetDirectionAwayFromNeighbour(CurrentBird, Neighbours);

// Alignment
if ( bEnableAlignment )
    AlignmentDirection = GetAlignmentDirection(CurrentBird, Neighbours);

// Projectiles
FVector ProjectileAvoid = GetProjectileAvoidDirection(CurrentBird);

// Waypoint
FVector ToWaypoint = CurrentBird.Waypoint - CurrentBird.Position;           // The vector from bird to waypoint
ToWaypoint.Normalize();

FVector Dir = ToWaypoint;                                         // The base direction is head to waypoint
Dir += AlignmentDirection;                                       // Add the direction change for alignment
Dir += DirectionToGroup;                                         // Add the direction change for grouping
Dir += DirectionAwayFromNeighbour;                                // Add the direction change for spacing
Dir += ProjectileAvoid;                                         // Add the direction change for avoiding projectiles

CurrentBird.Direction += Dir * 0.04f;    // Turn the bird

CurrentBird.Direction.Normalize();

// Add direction to current position * it's speed
CurrentBird.Position += CurrentBird.Direction * CurrentBird.Speed * DeltaTime;

if (FVector::Dist(CurrentBird.Position, CurrentBird.Waypoint) < 100.0f)          // If bird is within 5 metres of
waypoint...
{
    CurrentBird.Waypoint = PickNewWaypoint(CurrentBird.Waypoint);                  // Pick new waypoint.

}

CurrentBird.UpdateTransform();

```

```

        if ( bShowBirdsHeadings )
        {
            DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + CurrentBird.Direction * 60, 20.0f, FColor::Red, false, -1.0f, 0, 2.0f); // Current direction
            DrawDebugDirectionalArrow(GetWorld(), CurrentBird.Position, CurrentBird.Position + Dir * 80, 20.0f, FColor::Green, false, -1.0f, 0, 2.0f); // Desired heading
        }
    }

// ****
// ShowWaypoints - Draws a debug shpere around waypoint location
// ****
void AFlockingAI::ShowWaypoints()
{
    // Return if there are no birds yet...
    if ( !BirdList.Num() )
        return;

    // Loop through all waypoints and draw a sphere at its position in the world
    for (int i = 0; i < Waypoints.Num(); i++)
        DrawDebugSphere(GetWorld(), Waypoints[i]->GetActorLocation(), 30.0f, 8, FColor::Red, false,-1.0f, 0, 2.0f);
}

// ****
// PickNewWaypoint - Picks a new waypoint, ignoring the previous one
// ****
FVector AFlockingAI::PickNewWaypoint(FVector OldWaypoint)
{
    int Index = FMath::RandRange(0, Waypoints.Num() - 1); // Index is a random number between 0 and the number of waypoints minus 1.

    // If the distance between the old waypoint and the new waypoint
    // is less than 10, then skip it and get next waypoint along.
    if ( FVector::Dist(Waypoints[Index]->GetActorLocation(), OldWaypoint) < 10.0f )
        Index = (Index + 1) % Waypoints.Num(); // Get next index, making sure it stays within the bounds of the array
                                                // % (Modulo operator)
    divides left by right, then subtracts the remainder from this operation from the right.
    return Waypoints[Index]->GetActorLocation();
}

// ****
// GetNeighbours - Returns a list of pointers to neighbouring birds

```

```

// ****
TArray<FBird*> AFlockingAI::GetNeighbours(FBird& Bird)
{
    // Finds all neighbour birds for 'Bird'
    TArray<FBird*> Neighbours;                                // Container list to return
    for (int i = 0; i < BirdList.Num(); i++)
    {
        // Don't consider itself
        if (&BirdList[i] == &Bird)
            continue;

        // Distance check between 2 birds
        if (FVector::Dist(BirdList[i].Position, Bird.Position) < GroupingDistance)
            Neighbours.Add(&BirdList[i]);
    }
    return Neighbours;
}

// ****
// GetDirectionToGroup - Returns normalised direction to the centre of the group
// ****
FVector AFlockingAI::GetDirectionToGroup(FBird& CurrentBird, TArray<FBird*>& Neighbours)
{
    // Return value
    FVector DirectionToGroup(0.0f, 0.0f, 0.0f);

    // Make sure there are some birds
    if (Neighbours.Num())
    {
        // Vector for averaging the position of the birds in the group
        FVector GroupCentre(0.0f, 0.0f, 0.0f);

        for (int j = 0; j < Neighbours.Num(); j++)
            GroupCentre += Neighbours[j]->Position;           // Add all positions

        GroupCentre /= float(Neighbours.Num());                // divide by number of birds in group to get average position

        // Now get the vector from 'CurrentBird' to the group centre
        DirectionToGroup = GroupCentre - CurrentBird.Position;

        // Make sure it is noramlise (Unit Vector)
        DirectionToGroup.Normalize();
    }
}

```

```

        return DirectionToGroup;
    }

// *****
// GetDirectionAwayFromNeighbour - Returns the force vector to move 'CurrentBird' away from its neighbours
// *****
FVector AFlockingAI::GetDirectionAwayFromNeighbour(FBird& CurrentBird, TArray<FBird*>& Neighbours)
{
    // Return value
    FVector DirectionAwayFromNeighbour(0.0f, 0.0f, 0.0f);

    for (int j = 0; j < Neighbours.Num(); j++)
    {
        // Vector from neighbour to 'CurrentBird'
        FVector Direction = CurrentBird.Position - Neighbours[j]->Position;

        // Get the distance between the birds
        float Dist = Direction.Size();

        // Is it within the spacing distance?
        if (Dist <= SpacingDistance)
        {
            if (Dist > 0.001f)                                            // Prevent divide by zero (If
birds are in exactly the same place)
            {
                Direction /= Dist;                                         // Normalise
                float Weight = 1.0f - (Dist / SpacingDistance);           // Add more force the nearer we are to the
neighbouring bird (weighting)

                DirectionAwayFromNeighbour += Direction * Weight;         // Accumulate into return value
            }
        }
    }
    return DirectionAwayFromNeighbour;
}

// *****
// GetAlignmentDirection - Returns the force vector align 'CurrentBird' with it's neighbours
// *****
FVector AFlockingAI::GetAlignmentDirection(FBird& CurrentBird, TArray<FBird*>& Neighbours)
{
    // Return value
    FVector Alignment(0.0f, 0.0f, 0.0f);
}

```

```

// Make sure there are some birds
if ( Neighbours.Num() )
{
    // get the average of all neighbours directions
    for (int j = 0; j < Neighbours.Num(); j++)
        Alignment += Neighbours[j]->Direction;
    Alignment /= float(Neighbours.Num());
}
return Alignment;
}

// *****
// GetProjectileAvoidDirection - Returns direction away from any close projectiles
// *****
FVector AFlockingAI::GetProjectileAvoidDirection(FBird& CurrentBird)
{
    // Return value
    FVector ProjectileAvoid(0.0f, 0.0f, 0.0f);

    // Loop through all projectiles in the projectile list
    for (int p = 0; p < AFinalProjectRyanTProjectile::ProjectileList.Num(); p++)
    {
        // Look for projectiles that are close enough to avoid
        FVector ProjectilePos = AFinalProjectRyanTProjectile::ProjectileList[p]->GetActorLocation();
        if (FVector::Dist(CurrentBird.Position, ProjectilePos) < AvoidProjectileDistance)
        {
            // Get vector from bird to projectile
            FVector NewDir = ProjectilePos - CurrentBird.Position;
            NewDir.Normalize();

            // Check it;s type
            switch (AFinalProjectRyanTProjectile::ProjectileList[p]->Type )
            {
                case eProjectileType::Repel:
                    ProjectileAvoid -= NewDir;           // Move away from projectile
                    break;

                case eProjectileType::Attract:
                    ProjectileAvoid += NewDir;          // Move towards projectile
                    break;
            }
        }
    }
    return ProjectileAvoid;
}

```

```

}

// ****
// UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
// ****
void AFlockingAI::UpdateSubTitleText()
{
    if (TextContainer && TextContainerIndex >= 0 && TextContainerIndex < TextContainer->GetNumText())
    {
        TextContainer->SetTextFromList(-1);                                // Turn off 3d text

        FString Temp = TextContainer->GetTextFromList(TextContainerIndex);

        // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
        while (true)
        {
            int32 Index;
            if ( !Temp.FindChar( TCHAR('\r'), Index ) )
                break;
            Temp.RemoveAt( Index, 1 );
        }
        AFinalProjectRyanTHUD::SetString( Temp );
    }
}

// ****
// EnterVolume - Called when component enters collision volume
// ****
void AFlockingAI::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Create all birds
    CreateBirds();

    // Start the level sequence if we have one attached
    if ( LevelSequence )
    {
        FLevelSequencePlaybackSettings Settings;
        Settings.LoopCount = 0;

```

```

        SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
        SequencePlayer->Play();
    }

// *****
// EnterVolume - Called when component exits collision volume
// *****
void AFlockingAI::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
OtherBodyIndex)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Stop the level sequence
    if ( SequencePlayer )
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }

    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();

    // kill all birds
    DestroyBirds();
}

// *****
// CreateBirds - Creates all birds for the demonstration
// *****
void AFlockingAI::CreateBirds()
{
    // Number of birds to create
    int MaxBirds = 150;

    FBird Bird;

    for (int i = 0; i < MaxBirds; i++)
    {
        FVector Offset(FMath::FRandRange(-2000, 2000), FMath::FRandRange(-3000, 3000), FMath::FRandRange(-200, 200));

```

```

        Bird.Position = GetActorLocation() + Offset;           // Waypoint location plus rand offset
        Bird.Direction = FVector(-1, 0, 0);
        Bird.Waypoint = PickNewWaypoint(Bird.Position);       // Pick new waypoint.
        Bird.Speed = 250.0f;

        Bird.Speed += FMath::RandRange(-25, +25);

        // Create a skeletal mesh instance for this bird, using BirdMeshResource as a resource
        Bird.BirdMesh = NewObject<UStaticMeshComponent>(this, NAME_None);

        Bird.BirdMesh->SetStaticMesh(BirdMeshResource);

        Bird.UpdateTransform();
        Bird.BirdMesh->RegisterComponent();
        Bird.BirdMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);           // Don't need collision

        // KeepWorldTransform means that each component is not relative to the flockingAI actor, it has its own worldspace
position
        Bird.BirdMesh->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(),
FAttachmentTransformRules::KeepWorldTransform);

        LastAddedComponent = Bird.BirdMesh;

        BirdList.Add(Bird);
    }
}

// *****
// DestroyBirds - Destroys all birds
// *****
void AFlockingAI::DestroyBirds()
{
    // Empty the bird list, calls destructor for each element
    BirdList.Empty();
    LastAddedComponent = nullptr;
}

// *****
// OnEndSequenceEvent - Triggered when level sequence has finished playing
// *****
void AFlockingAI::OnEndSequenceEvent()
{

```

```

    // Stop the level sequence
    SequencePlayer->Stop();
    SequencePlayer = nullptr;

    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();
}

```

## Lighting

### *LightDemo.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "GameFramework/Actor.h"
#include "LevelSequence.h"
#include "RyanTCode/Misc/TextContainerComponent.h"
#include "LevelSequenceActor.h"
#include "Engine/Light.h"
#include "LightDemo.generated.h"

UCLASS()
class FINALPROJECTRYANT_API ALightDemo : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ALightDemo();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    // Pointer to the trigger volume the start/stop demo
    UPROPERTY(EditAnywhere)
    UBoxComponent* TriggerVolume = nullptr;
}

```

```

// Pointer to the container for all of demos text
UPROPERTY()
    UTTextContainerComponent* TextContainer = nullptr;

// Index of current text being displayed
UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|Text")
    float TextContainerIndex = -1; // -1 default = No Text

// Pointer to the attached level sequence
UPROPERTY(EditAnywhere, Category = "LightDemo|Trigger")
    ULevelSequence* LevelSequence = nullptr;

// Pointer to a light in the scene that demo relates to
UPROPERTY(EditAnywhere, Category = "LightDemo|Actors")
    ALight* Light = nullptr;

// Pointer to a camera in the scene that demo relates to
UPROPERTY(EditAnywhere, Category = "LightDemo|Actors")
    ACameraActor* Camera = nullptr;

// Pointer to the sequence player. Used to control the sequence after play has began. Pause etc...
UPROPERTY()
    ULevelSequencePlayer* SequencePlayer = nullptr;

// Bool that turns on/off drawing of the lights volume
UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|Debug")
    bool bShowLightRadius = false;

// The current AA mode
UPROPERTY(EditAnywhere, Interp, Category = "LightDemo|PostEffects")
    float AntiAliasMode = 2; // Has to be a float
not int to use UPROPERTY Interp. Otherwise it won't appear in timeline

UFUNCTION() // UFUNCTION tells the engine that this function can be used as a callback
(Required by Unreal)
    void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
UFUNCTION()
    void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
OtherBodyIndex);

private:
    void OnEndSequnceEvent();
    void UpdateSubTitleText();

```

```

    void DebugDrawLight();
    void SetAAMode(int Mode);
};

LightDemo.cpp
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "DrawDebugHelpers.h"
#include "FinalProjectRyanTHUD.h"
#include "FinalProjectRyanTCharacter.h"
#include "Components/DirectionalLightComponent.h"
#include "LightDemo.h"

// *****
// Constructor
// *****
ALightDemo::ALightDemo()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));

    RootComponent = TriggerVolume;
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void ALightDemo::BeginPlay()
{
    Super::BeginPlay();

    // Setup the trigger volume
    TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
    TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);

    // Register callbacks for enter/exit
    TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &ALightDemo::EnterVolume);
    TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &ALightDemo::ExitVolume);
}

```

```

// Get a list of all UTextContainerComponents
TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
if (TextComponents.Num())
    TextContainer = Cast<UTextContainerComponent>(TextComponents[0]);           // Use the first text container component
in list

    TextContainerIndex = -1;
}

// ****
// Tick - Called every frame
// ****
void ALightDemo::Tick( float DeltaTime )
{
    Super::Tick( DeltaTime );

    // Poll sequence player in it is playing to see if it has completed.
    if (SequencePlayer && !SequencePlayer->IsPlaying())
        OnEndSequnceEvent();

    UpdateSubTitleText();
    DebugDrawLight();

    if (SequencePlayer)
        SetAAMode(AntiAliasMode);           // Set AA Mode, but only if a sequence is playing
}

// ****
// EnterVolume - Called when component enters collision volume
// ****
void ALightDemo::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // Only allowed if overlapped by any kind of character
    if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
        return;

    // Get a pointer to the player character
    AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);

    // Exit of no player
    if (!Player)
        return;
}

```

```

// Start the level sequence playing
if (LevelSequence)
{
    FLevelSequencePlaybackSettings Settings;
    Settings.LoopCount = 0;
    SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
    SequencePlayer->Play();
}

// ****
// EnterVolume - Called when component exits collision volume
// ****
void ALightDemo::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
OtherBodyIndex)
{
    // Only allowed if overlapped by any kind of character
    if (!Other->GetClass()->IsChildOf(ACharacter::StaticClass()))
        return;

    // Get a pointer to the player character
    AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);

    // Exit of no player
    if (!Player)
        return;

    // Stop the level sequence
    if (SequencePlayer)
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }

    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();

    // Reset AA
    SetAAMode(2);
}

// ****

```

```

// OnEndSequnceEvent - Triggered when level sequence has finished playing
// ****
void ALightDemo::OnEndSequnceEvent()
{
    // Stop the level sequence
    if (SequencePlayer)
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }
    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();

    // Reset AA
    SetAAMode(2);
}

// ****
// UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
// ****
void ALightDemo::UpdateSubTitleText()
{
    if (TextContainer && TextContainerIndex >= 0 && TextContainerIndex < TextContainer->GetNumText() )
    {
        TextContainer->SetTextFromList(-1);                                // Turn off 3d text

        FString Temp = TextContainer->GetTextFromList(TextContainerIndex);

        // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
        while (true)
        {
            int32 Index;
            if (!Temp.FindChar(TCHAR('\r'), Index))
                break;
            Temp.RemoveAt(Index, 1);
        }
        AFinalProjectRyanTHUD::SetString(Temp);
    }
}

// ****
// UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
// ****

```

```

void ALightDemo::DebugDrawLight()
{
    if (!Light)
        return; // No light added so just return

    ULightComponent* LightComp = Light->GetLightComponent();

    /*****
    /* Point Lights */
    *****/
    if (LightComp->GetClass() == UPointLightComponent::StaticClass())
    {
        if (bShowLightRadius)
        {
            UPointLightComponent* Point = Cast<UPointLightComponent>(LightComp); // Cast from a light to a
point light

            // If it's a point light draw it's radius
            DrawDebugSphere(GetWorld(), Point->GetComponentLocation(), Point->AttenuationRadius, 32, FColor::White);
        }
    }

    /*****
    /* Spot Lights */
    *****/
    if (LightComp->GetClass() == USpotLightComponent::StaticClass())
    {
        if (bShowLightRadius)
        {
            USpotLightComponent* Spot = Cast<USpotLightComponent>(LightComp); // Cast from a light to a spot
light

            // If it's a spot light draw it's angled cones
            float OutAngle = FMath::DegreesToRadians(Spot->OuterConeAngle);
            float InAngle = FMath::DegreesToRadians(Spot->InnerConeAngle);

            DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X),
Spot->AttenuationRadius, OutAngle, OutAngle, 32, FColor::White);
            DrawDebugCone(GetWorld(), Spot->GetComponentLocation(), Spot->GetComponentTransform().GetUnitAxis(EAxis::X),
Spot->AttenuationRadius, InAngle, InAngle, 32, FColor::White);
        }
    }

    *****
}

```

```

/* Directional Lights */
/*****
if (LightComp->GetClass() == UDirectionalLightComponent::StaticClass())
{
    if (bShowLightRadius)
    {
        UDirectionalLightComponent* DirLight = Cast<UDirectionalLightComponent>(LightComp); // Cast
from a light to a directional light
        FVector ArrowEnd = DirLight->GetComponentLocation() + DirLight->GetDirection() * 180.0f; // The arrow
will be 180 world units long
        DrawDebugDirectionalArrow(GetWorld(), DirLight->GetComponentLocation(), ArrowEnd, 40.0f, FColor::White,
false, 0.0f, 0, 3.0f);
    }
}

// ****
// SetAAMode - Updates AntiAlias mode
// ****
void ALightDemo::SetAAMode(int Mode)
{
    // run a console command to change the AA mode

    //    0 : off
    //    1 : FXAA
    //    2 : TemporalAA
    //    3 : MSAA

    FString Command = "r.DefaultFeature.AntiAliasing ";
    Command.AppendInt(Mode);
    GetWorld()->GetFirstPlayerController()->ConsoleCommand(Command, false);
}

```

## Miscellaneous

### *MyStaticMesh.h*

```
// Copyright © 2018 Ryan Tinman, All Rights Reserved.
```

```
#pragma once
```

```
#include "Engine/StaticMeshActor.h"
#include "MyStaticMesh.generated.h"
```

```

/**
 *
 */
UCLASS()
class FINALPROJECTRYANT_API AMyStaticMesh : public AStaticMeshActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AMyStaticMesh();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

    // Called every frame
    virtual void Tick(float DeltaSeconds) override;

    // The radius to use for simple sphere collision checks
    UPROPERTY(EditAnywhere, Category = "MyStaticMesh Settings")
        float SphereRadius = 52.0f;

    // A static list that contains all instances of AMyStaticMesh. Used in VerletSim class for collisions
    static TArray<AMyStaticMesh*> MeshList;
};


```

### *MyStaticMesh.cpp*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "DrawDebugHelpers.h"
#include "MyStaticMesh.h"

TArray<AMyStaticMesh*> AMyStaticMesh::MeshList;           // Static list that holds all 'AMyStaticMesh' instances. Used by
collision.

// *****
// Constructor
// *****
AMyStaticMesh::AMyStaticMesh()

```

```

{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
}

// ****
// Tick - Called every frame
// ****
void AMyStaticMesh::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

// ****
// BeginPlay - Called when the game starts or when spawned
// ****
void AMyStaticMesh::BeginPlay()
{
    Super::BeginPlay();

    // Add this mesh to the static mesh list
    MeshList.Add(this);
}

// ****
// EndPlay - Called when the game ends or when deleted
// ****
void AMyStaticMesh::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    Super::EndPlay(EndPlayReason);

    // Remove this mesh from the static mesh list
    MeshList.Remove(this);
}

```

### *TextContainerComponent.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "Components/TextRenderComponent.h"
#include "TextContainerComponent.generated.h"

```

```

/*
 *
 */

// UCLASS specifiers copied from base class (UTextRenderComponent)
UCLASS(ClassGroup=Rendering, hideCategories=(Object,LOD,Physics,TextureStreaming,Activation,"Components|Activation",Collision),
editinlinenew, meta=(BlueprintSpawnableComponent = ""))
class FINALPROJECTRYANT_API UTextContainerComponent : public UTextRenderComponent
{
    GENERATED_BODY()

public:

    bool SetTextFromList(int Index);

    // Returns a the string at the index in the text list
    FString GetTextFromList(int Index)
    {
        if (Index < 0 || Index >= TextList.Num())
            return FString(); // Return an empty string
        return TextList[Index];
    }

    // Returns the number of texts in the list
    int GetNumText() { return TextList.Num(); }

private:

    // List of strings that can be displayed
    UPROPERTY(EditAnywhere, Category = "TextList", meta = (MultiLine = true)) // MultiLine allows line feeds to be added
    in details panel using "Shift-Enter"
    TArray<FString> TextList;
};


```

### *TextContainerComponent.cpp*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "TextContainerComponent.h"

```

```

// ****
// SetTextFromList - Set current text from list of strings that can be displayed
// ****
bool UTextContainerComponent::SetTextFromList( int Index )
{
    // Set the current text from n index into text klist

    if ( Index < 0 || Index >= TextList.Num() )           // Bounds checking
    {
        SetText(FText::FromString(""));
        return false;
    }

    SetText( FText::FromString( *TextList[Index] ) );      // Convert FString from array to FText needed by
UTextRenderComponent::SetText()
    return true;
}

```

## Molecular Simulation

*MolecularSim.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "GameFramework/Actor.h"
#include "MolecularSim.generated.h"

UCLASS()
class FINALPROJECTTRYANT_API AMolecularSim : public AActor
{
    GENERATED_BODY()

public:

    struct Particle
    {
        FVector Pos;
        FVector PreviousPos;
        FVector ExternalForce;

```

```

UPROPERTY()
collection
UStaticMeshComponent* Mesh = nullptr; // Tag Mesh as a UPROPERTY so that Unreal garbage
// knows it exists and will need deleting when not referenced

float Radius = 0.0f;
int CellIndex = -1;
int ParticleIndex = -1;

void AddForce(FVector Force)
{
    ExternalForce += Force;
}

void UpdateScale()
{
    Mesh->SetRelativeScale3D( FVector( Radius * (2.0f/100.0f) ) ); // (Diameter = R * 2) (Normalise sphere
size = Divide by 100. Assumes geometry has 1m radius)
}

void UpdateTransform()
{
    if (!Mesh) // Make sure we have a mesh instance before updating transform
        return;

#ifndef _MSC_VER
    float s = Radius * (2.0f/100.0f); // (Diameter = R * 2) (Normalise sphere size = Divide by 100.
Assumes geometry has 1m radius)
    FTransform Trans;
    Trans.SetLocation(Pos);
    Trans.SetScale3D( FVector(s) );
    Mesh->SetWorldTransform(Trans); // This is extremely slow
#else
    Mesh->SetRelativeLocationAndRotation(Pos, FQuat()); // This is extremely slow
    Mesh->SetRelativeScale3D( FVector(s) );
#endif
}
};

// Sets default values for this actor's properties
AMolecularSim();

protected:
    // Called when the game starts or when spawned

```

```

virtual void BeginPlay() override;

// Cleanup stuff
virtual void EndPlay(const EEndPlayReason::Type Reason) override;

private:

void CreateParticleCells();
void UpdateParticleCell( AMolecularSim::Particle& P );
void CreateParticles();
void DebugDrawParticleCells();
void Simulate( float DeltaTime );
void UpdateTransforms();
void CreateParticle( FVector Position, float Radius );

public:
// Called every frame
virtual void Tick(float DeltaTime) override;

UPROPERTY( EditAnywhere )
float Mass = 0.01f; // Mass of a particle

UPROPERTY( EditAnywhere )
float Damping = 0.5f; // Damping for the Verlet intergration

UPROPERTY( EditAnywhere )
float AttractDist = 50.0f; // Distance be 2 particles are attracted to each other

UPROPERTY( EditAnywhere )
float AttractionWeight = 0.2f; // Weighting for attraction as a percentage (0.0 - 1.0)

UPROPERTY(EditAnywhere)
UStaticMesh* ParticleMeshResource = nullptr;

UPROPERTY(EditAnywhere)
UBoxComponent* ParticleInitVolume = nullptr;

UPROPERTY(EditAnywhere)
UBoxComponent* SimulationVolume = nullptr;

UPROPERTY(EditAnywhere)
float ParticleRadius = 10.0f;

TArray<Particle> Particles;

```

```

float Timer = 0.0f;

float CellSize = 80.0f; // Cell size in world units

FVector MinBound; // Min world bounds of simulate volume
FVector MaxBound; // Max world bounds of simulate volume

int CellsX = 0; // Dimensions in cells
int CellsY = 0;
int CellsZ = 0;

int NumCells = 0; // Total number of cells in CellList
(CellsX*CellsY*CellsZ)

struct ParticleCell // Structure to define a single cell (particle
container)
{
    FVector DebugPos;
    TArray <int> ParticleIndices; // Index list into Particles list
};

TArray <ParticleCell> Cells; // The list for all cells

private:
    UPrimitiveComponent* LastAddedComponent = nullptr;
};


```

### *MolecularSim.cpp*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```

#include "FinalProjectRyanT.h"
#include "Engine.h"
#include "DrawDebugHelpers.h"
#include "MolecularSim.h"

DECLARE_CYCLE_STAT(TEXT("Molecular Sim: Simulate"), STAT_MolecularSim, STATGROUP_Ryan); // Stat within group 'Ryan' See
"FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Molecular Sim: Update Transform"), STAT_UpdateTransform, STATGROUP_Ryan);

#define DISABLE // Put this in to disable simulation, rem out to enable

// *****

```

```

// Constructor
// *****
AMolecularSim::AMolecularSim()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));

    // UBoxComponent is used to create a volume around whole simulation system. Must stay axis aligned so that cell checks are faster.
    SimulationVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("SimulationVolume"));
    SimulationVolume->SetCollisionEnabled( ECollisionEnabled::NoCollision );

    // UBoxComponent is used to create a volume where the particles are initially created in an even grid.
    ParticleInitVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("ParticleInitVolume"));
    ParticleInitVolume->SetCollisionEnabled( ECollisionEnabled::NoCollision );
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void AMolecularSim::BeginPlay()
{
    Super::BeginPlay();

#ifdef DISABLE
    return;
#endif

    CreateParticleCells();           // Create the cells
    CreateParticles();              // Create the particles in the init volume
}

// *****
// EndPlay - Cleanup stuff
// *****
void AMolecularSim::EndPlay(const EEndPlayReason::Type Reason)
{
    Super::EndPlay(Reason);

    Particles.Empty();
    Cells.Empty();
}

```

```

// *****
// Tick - Called every frame
// *****
void AMolecularSim::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

#ifndef DISABLE
    return;
#endif

    FString s = "*** Simulating ";
    s.AppendInt( Particles.Num() );
    s.Append( " Molecules **" );
    GEngine->AddOnScreenDebugMessage(-1, 0.0f, FColor::Red, s);

    DebugDrawParticleCells();
    Simulate(DeltaTime);
    UpdateTransforms();
}

// *****
// AMolecularSim - Simulate physics for all particles
// *****
void AMolecularSim::Simulate( float DeltaTime )
{
    SCOPE_CYCLE_COUNTER(STAT_MolecularSim);

    Timer += DeltaTime;

    bool bCanMove = Timer > 3.0f;

    // Attract Dist (Or Max Dist we need to consider)
    float R = (ParticleRadius * 2.5f) * 2.0f;

//bCanMove = false;

    if ( bCanMove )
    {
        for ( int i = 0 ; i < Particles.Num() ; i++ )
        {
            Particle& P = Particles[i];

```

```

P.AddForce( FVector(0.0f, 0.0f, -9.80f) );                                // Add gravity

#ifndef _PARTICLE_H_
// TODO: Change this to be cell based. Check collision for just neighbouring cells will be much faster...
// Check for collision against other particles
for ( int j = 0 ; j < Particles.Num() ; j++ )
{
    if ( i == j )
        continue;

    Particle& P2 = Particles[j];

    if ( P.CellPos != P2.CellPos )
        continue;

    FVector Dir = P2.Pos - P.Pos;                                         // Direction From P to P2

    float Dist = Dir.Size();
    if ( Dist > ParticleRadius )
        continue;

    Dir /= Dist;               // Normalise

    // Attraction (To simulate viscosity)
    if ( Dist <= AttractDist )
    {
        float Weight = 1.0f - (Dist / AttractDist);
        P.AddForce( Dir * Weight * AttractionWeight );
    }

    // Collision
    if ( Dist <= (P.Radius+P2.Radius) )
    {
        FVector V1 = ( P.Pos- P.PreviousPos );
        FVector V2 = (P2.Pos-P2.PreviousPos);

        // Set new acceleration for both
        P.AddForce(V2);
        P2.AddForce(V1);

        // Re-position both and zero current velocity
        P.PreviousPos = P.Pos = P2.Pos - (Dir * (P.Radius+P2.Radius));
        P2.PreviousPos = P2.Pos = P.Pos + (Dir * P.Radius);
    }
}

```

```

        //Acc = FVector::ZeroVector;
    }

#endif // 0

// Apply Verlet integration
FVector Acc = P.ExternalForce / Mass;
P.Pos += ((P.Pos-P.PreviousPos)*Damping) + (Acc*DeltaTime);

// Simple bounds check to keep all particles within the simulation volume
if ( P.Pos.X < MinBound.X ) { P.PreviousPos.X = P.Pos.X = MinBound.X; }
if ( P.Pos.Y < MinBound.Y ) { P.PreviousPos.Y = P.Pos.Y = MinBound.Y; }
if ( P.Pos.Z < MinBound.Z ) { P.PreviousPos.Z = P.Pos.Z = MinBound.Z; }

if ( P.Pos.X > MaxBound.X ) { P.PreviousPos.X = P.Pos.X = MaxBound.X; }
if ( P.Pos.Y > MaxBound.Y ) { P.PreviousPos.Y = P.Pos.Y = MaxBound.Y; }
//if ( P.Pos.Z > MaxBound.Z ) { P.PreviousPos.Z = P.Pos.Z = MaxBound.Z; }           // No Top Check (Fish Tank)

P.ExternalForce = FVector::ZeroVector;

#if 0
// Some test Unreal collision using raytrace
FHitResult HitResult;
FCollisionQueryParams Params;
FCollisionObjectQueryParams ObjParams;

ObjParams.AddObjectTypeToQuery( ECC_WorldStatic );
ObjParams.AddObjectTypeToQuery( ECC_WorldDynamic );

FVector Dir = (P.Pos-P.PreviousPos);
Dir.Normalize();
Dir *= P.Radius;

if (GetWorld()->LineTraceSingleByObjectType( HitResult, P.PreviousPos+Dir, P.Pos+Dir, ObjParams, Params))
{
    float Speed = FVector::Dist( P.Pos, P.PreviousPos );
    FVector NewPos = HitResult.ImpactPoint + (HitResult.ImpactNormal*P.Radius);

    FVector Force = HitResult.ImpactNormal * Speed;

    P.Pos          = NewPos;
    P.PreviousPos = NewPos;
    P.AddForce( Force );
}

```

```

#endif
    P.PreviousPos = P.Pos;
    UpdateParticleCell( P );
}
}

// ****
// UpdateTransforms - Update all particle model transforms from their positions
// ****
void AMolecularSim::UpdateTransforms()
{
    SCOPE_CYCLE_COUNTER(STAT_UpdateTransform);

    // Loop through all particles
    for ( int i = 0 ; i < Particles.Num() ; i++ )
    {
        Particles[i].UpdateTransform();                                // Update it's transform
    }
}

// ****
// CreateParticleCells - Builds the cell list array
// ****
void AMolecularSim::CreateParticleCells()
{
    // Get the extents of the 'Simulation' volume
    FVector VolumeExtents = SimulationVolume->GetScaledBoxExtent();
    FVector VolPos = SimulationVolume->GetComponentLocation();
    MinBound = VolPos - VolumeExtents;
    MaxBound = VolPos + VolumeExtents;
    VolumeExtents *= 2.0f;

    // Divide the world space volume into number of cells needed
    CellsX = 1 + FMath::CeilToInt( VolumeExtents.X / CellSize );
    CellsY = 1 + FMath::CeilToInt( VolumeExtents.Y / CellSize );
    CellsZ = 1 + FMath::CeilToInt( VolumeExtents.Z / CellSize );

    NumCells = CellsX*CellsY*CellsZ;           // Total number of cells

    // Allocate the cells
    for ( int z = 0 ; z < CellsZ ; z++ )
    {

```

```

        for ( int y = 0 ; y < CellsY ; y++ )
        {
            for ( int x = 0 ; x < CellsX ; x++ )
            {
                ParticleCell Cell;
                Cell.DebugPos = MinBound + FVector( float(x)*CellSize, float(y)*CellSize, float(z)*CellSize );
                Cells.Add(Cell);
            }
        }
    }

// ****
// UdateParticleCell - Update a single particles cell
// ****
void AMolecularSim::UpdateParticleCell( AMolecularSim::Particle& P )
{
    FVector RelPos = P.Pos - MinBound;

    // Calculate the cell x, y, z index into the Cell array
    int x = FMath::FloorToInt( RelPos.X / CellSize );
    int y = FMath::FloorToInt( RelPos.Y / CellSize );
    int z = FMath::FloorToInt( RelPos.Z / CellSize );

    // Do some bounds checking. Don't update if out of bounds
    if ( x < 0 || x >= CellsX )
        return;

    if ( y < 0 || y >= CellsY )
        return;

    if ( z < 0 || z >= CellsZ )
        return;

    // Calculate the index into the cell list from cell x, y, z
    int NewCell = x + (y*CellsX) + (z*(CellsX*CellsY));

    // If it is already in this cell just return
    if ( P.CellIndex == NewCell )
        return;

    // Remove from old cell
    if ( P.CellIndex >= 0 )                                // If the it's index if < 0 then it's
never been added to the cell list before

```

```

        Cells[P.CellIndex].ParticleIndices.Remove(P.ParticleIndex);

        // Set the particles new cell index (Index into CellList)
        P.CellIndex = NewCell;

        // Add to new cell
        Cells[NewCell].ParticleIndices.Add(P.ParticleIndex);
    }

// *****
// DebugDrawParticleCells - Draw each cell bounding box if it contains any particles
// *****
void AMolecularSim::DebugDrawParticleCells()
{
    // Allocate the cells
    for ( int i = 0 ; i < NumCells ; i++ )
    {
        ParticleCell& Cell = Cells[i];

        // Draw This Cells Volume
        if ( Cell.ParticleIndices.Num() )
        {
            FVector Centre = Cell.DebugPos + FVector(CellSize*0.5f,CellSize*0.5f,CellSize*0.5f);
            DrawDebugBox(GetWorld(), Centre, FVector(CellSize*0.5f,CellSize*0.5f,CellSize*0.5f), FColor::Green );
        }
    }

    // Draw The System Volume
    DrawDebugBox(GetWorld(), SimulationVolume->GetComponentLocation(), SimulationVolume->GetScaledBoxExtent(), FColor::Cyan );

    // Draw The Particle Init Volume
    DrawDebugBox(GetWorld(), ParticleInitVolume->GetComponentLocation(), ParticleInitVolume->GetScaledBoxExtent(), FColor::Red
);
}

// *****
// CreateParticles - Create all particles within the ParticleInitVolume
// *****
void AMolecularSim::CreateParticles()
{
    float Step = ParticleRadius * 2.0f;                                // Initialise then all 'particle diameter' apart...

    // Get the world space extents of the init volume
    FVector VolumeExtents = ParticleInitVolume->GetScaledBoxExtent();
}

```

```

FVector VolPos = ParticleInitVolume->GetComponentLocation();

// Get the maximum/minimum positions
FVector MinB = VolPos - VolumeExtents;
FVector MaxB = VolPos + VolumeExtents;

// Loop through and create a particle at each position in the volume
for ( float z = MinB.Z ; z < MaxB.Z ; z += Step )
{
    for ( float y = MinB.Y ; y < MaxB.Y ; y += Step )
    {
        for ( float x = MinB.X ; x < MaxB.X ; x += Step )
        {
            CreateParticle( FVector( x, y, z ), ParticleRadius );
        }
    }
}

UE_LOG(LogTemp, Warning, TEXT("## Simulating %d Particles **"), Particles.Num() );
}

// *****
// CreateParticle - Create a single particle and add it to Particle list
// *****
void AMolecularSim::CreateParticle( FVector Position, float Radius )
{
    Particle P;
    P.Pos           = Position;
    P.PreviousPos = P.Pos;
    P.Radius        = Radius;
    P.ExternalForce = FVector(0,0,0);

    // Add a mesh component if there is a resource for it.
    if ( ParticleMeshResource )
    {
        // Create a static mesh instance for this particle, using ParticleMeshResource as a resource

        P.Mesh = NewObject<UStaticMeshComponent>(this, NAME_None);

        P.Mesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);

        P.Mesh->SetStaticMesh(ParticleMeshResource);

        P.Mesh->RegisterComponent();
    }
}

```

```

        FTransform Trans;
        Trans.SetLocation(Position);
        P.Mesh->SetWorldTransform(Trans);

    //    P.UpdateTransform();
    //    P.UpdateScale();
}

// Add this particle to the particle list and get the index it was added at.
int AddedAt = Particles.Add(P);

// Once added, set it's index into the list. This is used for addition into cells etc.
P.ParticleIndex = Particles[AddedAt].ParticleIndex = AddedAt;

// Now update this particles cell info
UpdateParticleCell( Particles[AddedAt] );
}

```

## Physics

### *RopeSim.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "VerletSim.h"
#include "RopeSim.generated.h"

#define MAX_ROPE_SEGMENTS 30

UCLASS()
class FINALPROJECTRYANT_API ARopeSim : public AVerletSim
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ARopeSim();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

```

```

// Called every frame
virtual void Tick(float DeltaSeconds) override;

// How thick the rope mesh is
UPROPERTY(EditAnywhere, Category = "Verlet Settings", meta = (ClampMin = 0.1f, ClampMax = 100.0f ))
float RopeThickness = 5.0f;

// Number of segments around the rope mesh
UPROPERTY(EditAnywhere, Category = "Verlet Settings", meta = (ClampMin = 3, ClampMax = 30 )) // If Max changed, also
change MAX_ROPE_SEGMENTS above to match
int RopeSegments = 8;

private:
    void Constrain();
    void Simulate(float DeltaTime);
    void Render();
    void ResolveCollision();
    void DebugDraw(int i);
    void BuildMesh();
    void BuildConstraints();
    void UpdateMesh();
    void UpdateLockedVerts();

    // Number of verts in the rope
    static const int NumVerts = 35;

    // Linear vertex list
    Verlet Verts[NumVerts];

};


```

### RopeSim.cpp

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```

#include "FinalProjectRyanT.h"
#include "RopeSim.h"
#include "DrawDebugHelpers.h"
#include "Kismet/GameplayStatics.h"
#include "SceneManagement.h"

// For UE4 Profiler: Type stat Ryan in console to see timing stats
DECLARE_CYCLE_STAT(TEXT("Rope Constrain") , STAT_RopeConstrain , STATGROUP_Ryan); // Stat within
group 'Ryan' See "FinalProjectRyanT.h"

```

```

DECLARE_CYCLE_STAT(TEXT("Rope UpdateMesh")) , STAT_RopeUpdateMesh , STATGROUP_Ryan); // Stat within
group 'Ryan' See "FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Rope ResolveCollision")) , STAT_RopeResolveCollision , STATGROUP_Ryan); // Stat within group
'Ryan' See "FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Rope Simulate")) , STAT_RopeSimulate , STATGROUP_Ryan); // Stat within
group 'Ryan' See "FinalProjectRyanT.h"

// ****
// Constructor
// ****
ARopeSim::ARopeSim()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
}

// ****
// BeginPlay - Called when the game starts or when spawned
// ****
void ARopeSim::BeginPlay()
{
    Super::BeginPlay();

    // Initialising all verts in rope

    FVector Pos = GetActorLocation(); // Get the position of this ARopeSim actor instance

    for (int i = 0; i < NumVerts; i++)
    {
        Verts[i].Flags = 0; // Reset the flags
        Verts[i].Pos = Pos; // Set the position
        Verts[i].PreviousPos = Pos; // Set the previous position to the same
        Pos.Z -= EdgeLength; // Move down the rope to the next vert
    }

    BuildConstraints(); // Build the constraint list
    BuildMesh(); // Construct the mesh using the above initialised verts
}

// ****
// Tick - Called every frame
// ****
void ARopeSim::Tick(float DeltaTime)
{
}

```

```

Super::Tick(DeltaTime);

// Check if we are close enough to tick (Optimisation)
if ( OutsideOfTickRange() )
    return;

UpdateLockedVerts();                                // Update any verts that are locked
Simulate(DeltaTime);                               // Tick the simulation for all vert

// Constraint loop to iron out any inconsistencies caused by collision/constrain 'pushing' and 'pulling' the verts
for (int i = 0; i < NumConstrainIterations; i++)
{
    Constrain();
    ResolveCollision();
}

Render();                                         // Update the mesh
}

// ****
// UpdateLockedVerts - Update any verts that are locked
// ****
void ARopeSim::UpdateLockedVerts()
{
    // Clear all flags (Allows real-time editing of attach points)
    for (int i = 0; i < NumVerts; i++)
        Verts[i].Flags = 0;

    FTransform Transform = GetActorTransform();

    for (int i = 0 ; i < AttachmentList.Num() ; i++ )
    {
        int Index = AttachmentList[i].VertIndex;

        if ( Index >= 0 && Index < NumVerts )
        {
            FVector Pos = AttachmentList[i].AttachPos;
            if (AttachmentList[i].bRelativeToActor)
                Pos = Transform.TransformPosition(Pos);           // Transform into World Space as Attachment List]
positions are in local space

            Verts[Index].Pos = Pos;
            Verts[Index].PreviousPos = Pos;
        }
    }
}

```

```

        Verts[Index].Flags = 1;                                // Tell the simulation not to simulate for
this vert or move during constraining
    }
}
}

// *****
// ResolveCollision - Resolve collisions for all verts (Based on the velocity)
// *****
void ARopeSim::ResolveCollision()
{
    if ( !bEnableCollision )
        return;

    SCOPE_CYCLE_COUNTER(STAT_RopeResolveCollision);           // For UE4 profiling

    for (int i = 0; i < NumVerts; i++)
    {
        // Resolve collision for current vert
        if (ResolveVertCollision(Verts[i]))
        {
            // Listener for rope debug draw tickbox
            if (bEnableDebugDraw)
                DebugDraw(i);
        }
    }
}

// *****
// Constrain - Maintain equal distance between all verts
// *****
void ARopeSim::Constrain()
{
    SCOPE_CYCLE_COUNTER(STAT_RopeConstrain);                  // For UE4 profiling

    for (int i = 0; i < ConstraintList.Num() ; i++)
    {
        ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
    }
}

// *****
// Simulate - Simulate verlet physics for rope
// *****

```

```

void ARopeSim::Simulate(float DeltaTime)
{
    SCOPE_CYCLE_COUNTER(STAT_RopeSimulate);           // For UE4 profiling

    for (int i = 0; i < NumVerts; i++)
        SimulateVert(Verts[i], DeltaTime);
}

// ****
// Render - Renders the rope
// ****
void ARopeSim::Render()
{
    UpdateMesh();
    return;

#if 0
    // For each vert, draw a debug line, alternating between the colours red and blue.
    for (int i = 0; i < NumVerts - 1; i++)
    {
        float Length = (Verts[i].Pos - Verts[i + 1].Pos).Size();

        FColor Colour = (i & 1) ? FColor::Red : FColor::Blue;

        //if (fabs(Length - EdgeLength) > 0.1f)
        //    Colour = FColor::Black;

        DrawDebugLine(GetWorld(), Verts[i].Pos, Verts[i + 1].Pos, Colour, false, -1.0f, 0, 5.0f);
    }
#endif
}

// ****
// DebugDraw - Draws a sphere at the position of vertex index 'i'
// ****
void ARopeSim::DebugDraw(int i)
{
    DrawDebugSphere(GetWorld(), Verts[i].Pos, 10.0f, 8, FColor::White);
}

// ****
// BuildMesh - Build the textured mesh
// ****

```

```

void ARopeSim::BuildMesh()
{
    MeshComponent->ClearAllMeshSections();                                // Clear any old meshes

    TArray <FVector>             Vertices;
    TArray <int32>               Triangles;
    TArray <FVector>             Normals;
    TArray <FVector2D>           UV0;
    TArray <FColor>              VertexColor;
    TArray <FPProcMeshTangent>   VertexCTangent;

    // Build the vertex list

    for (int i = 0; i < NumVerts; i++)
    {
        float Rad = 0.0f;
        float RadDelta = (PI*2.0f) / float(RopeSegments);      // Circumference divided by the number of segments (In
    Radians)
        for ( int s = 0 ; s < RopeSegments ; s++ )
        {
            FVector Offset;
            Offset.X = cosf( Rad ) * RopeThickness;
            Offset.Y = sinf( Rad ) * RopeThickness;
            Offset.Z = 0.0f;                                     // Always on horizontal plane

            FVector Pos = (Verts[i].Pos + Offset);

            Vertices.Add( Pos );

            FVector Normal = Offset;
            Normal.Normalize();
            Normals.Add( Normal );

            FVector2D UV = FVector2D(s,i);
            UV.X /= float(RopeSegments);                         // Horizontal: 0.0 to 1.0 mapped around the
            rope
            bottom
            UV.Y /= float(NumVerts);                            // Vertical: 0.0 to 1.0 from top of rope to

            UV *= UVTileScale;

            UV0.Add(UV);

            Rad += RadDelta;                                    // Next segment in radians
        }
    }
}

```

```

        }

    }

    // Build the triangle list. Each trianlge has 3 x Indices into the above vertex list.
    //
    //      0----1
    //      | \   |
    //      |  \  |
    //      |   \ |
    //      |    \| |
    // RopeSegments+0----RopeSegments+1
    //
    for (int i = 0; i < NumVerts-1; i++)
    {
        for ( int s = 0 ; s < RopeSegments ; s++ )
        {
            int BaseOffset = (i*RopeSegments) + s;

            Triangles.Add( BaseOffset + 0 );
            Triangles.Add( BaseOffset + RopeSegments );           // +RopeSegments moved down the rope to the next verts
segment verts
            Triangles.Add( BaseOffset + RopeSegments + 1 );

            Triangles.Add( BaseOffset + RopeSegments+1 );
            Triangles.Add( BaseOffset + 1 );
            Triangles.Add( BaseOffset + 0 );
        }
    }

    bool bCreateCollision = false;
    MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );

    MeshComponent->SetMaterial(0,MeshMaterial);

    MeshComponent->SetWorldTransform( FTransform() );           // Set transform to ID (Zero Position & Zero Rotation) as all
verts are in world space.
}

// ****
// UpdateMesh - Update the textured mesh
// ****
void ARopeSim::UpdateMesh()
{
    SCOPE_CYCLE_COUNTER(STAT_RopeUpdateMesh);                // For UE4 profiling
}

```

```

TArray < FVector > Vertices;
TArray < FVector > Normals;
TArray < FVector2D > UV0;
TArray < FColor > VertexColor;
TArray < FProcMeshTangent > VertexCTangent;

FVector SurfaceNormals[NumVerts*MAX_ROPE_SEGMENTS];

for (int i = 0; i < NumVerts; i++)
{
    FVector Up, Side, Forward;

    // Get the vector pointing up the rope at this vert
    if ( i < NumVerts-1 )
        Up = Verts[i].Pos - Verts[i+1].Pos;
    else
        Up = Verts[i-1].Pos - Verts[i].Pos;
    Up.Normalize();

    Side = FVector::CrossProduct( Up, FVector(0,1,0) ); // Get the side vector (Perpendicular to the world Y Axis
(Forward) )
    Forward = FVector::CrossProduct( Up, Side ); // Get the forward vector (Perpendicular Side
axis and Up axis)
    Side = FVector::CrossProduct( Up, Forward ); // Get the 'real' side vector (Perpendicular to Up and
Forward)

    float Rad = 0.0f;
    float RadDelta = (PI*2.0f) / float(RopeSegments); // Circumference divided by the number of segments (In
Radians)
    for (int s = 0; s < RopeSegments; s++)
    {
#if 0
        FVector Offset;
        Offset.X = cosf(Rad) * RopeThickness;
        Offset.Y = sinf(Rad) * RopeThickness;
        Offset.Z = 0.0f;
        // Always on horizontal plane
#else
        FVector Offset = Side * cosf(Rad);
        Offset += Forward * sinf(Rad);

        Offset.Normalize();
#endif
        FVector Pos = (Verts[i].Pos + Offset * RopeThickness);
    }
}

```

```

        Vertices.Add(Pos);

        // Store surface normal for next stage, vertex normals
        SurfaceNormals[(i*RopeSegments)+s] = Offset;

//        DrawDebugLine(GetWorld(), Pos, Pos + SurfaceNormals[(i*RopeSegments)+s] * 20.0f, FColor::Cyan, false, -1.0f,
0, 2.0f);

        Rad += RadDelta;                                // Next segment in radians
    }

}

// Build the vertex normals from the surface normals
for (int i = 0; i < NumVerts; i++)
{
    float Rad = 0.0f;
    float RadDelta = (PI*2.0f) / float(RopeSegments);           // Circumference divided by the number of segments (In
Radians)
    for (int s = 0; s < RopeSegments; s++)
    {
        FVector Normal(0.0f,0.0f,0.0f);

        if ( i > 0 )
            Normal += SurfaceNormals[((i-1)*RopeSegments)+s];
        if ( i < (NumVerts-1) )
            Normal += SurfaceNormals[((i+1)*RopeSegments)+s];

        if ( s > 0 )
            Normal += SurfaceNormals[(i*RopeSegments)+(s-1)];
        if ( s < (RopeSegments-1) )
            Normal += SurfaceNormals[(i*RopeSegments)+(s+1)];

        Normal.Normalize();
        Normals.Add( Normal );
    }

    //FVector Pos = Vertices[(i*RopeSegments)+s] + GetActorLocation();
    //DrawDebugLine(GetWorld(), Pos, Pos + Normal * 20.0f, FColor::Cyan, false, -1.0f, 0, 2.0f);
}
}

MeshComponent->UpdateMeshSection( 0, Vertices, Normals, UV0, VertexColor, VertexCTangent );
}

```

```

// ****
// BuildConstraints - Build constraints for all verts
// ****
void ARopeSim::BuildConstraints()
{
    // Build a list of constraint that run down the rope
    for (int i = 0; i < NumVerts-1; i++)
        CreateConstraint(&Verts[i], &Verts[i+1], eVerletConstraintType::Normal );
}

```

### *VerletSim.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "GameFramework/Actor.h"
#include "RyanTCode/Wind/WindSim.h"
#include "ProceduralMeshComponent.h"
#include "RyanTCode/Misc/TextContainerComponent.h"
#include "LevelSequence.h"
#include "LevelSequenceActor.h"
#include "VerletSim.generated.h"

UENUM()
details panels
enum class eVerletConstraintType : uint8
{
    All,
    Normal,
    Diagonal,
    NormalSeconary,
    DiagonalSeconary
};

UENUM()
enum class eRenderMode : uint8
{
    None,
    WireframeSquares,
    WireframeTriangles,
    WireframeConstraints,
    WireframeSquaresAndConstraints,

```

```

ProgressiveMesh,
ProgressiveMeshSquares,
ProgressiveMeshTriangles
};

UENUM()
enum class eShowForceMode : uint8
{
    None,
    ShowWithMagnitude,
    ShowFixedLength,
    ShowWithModulation
};

// ****
// FVerletAttachment - For adding dynamic attach/lock points to rope/cloth
// ****
USTRUCT() // Tell Unreal Editor that this structure is exposed for editing in
its details panels
struct FVerletAttachment
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, Category = "Verlet Settings")
        bool bRelativeToActor = true;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings", Meta = (MakeEditWidget = true))
        FVector AttachPos = FVector(0.0f, 0.0f, 0.0f);

    UPROPERTY(EditAnywhere, Category = "Verlet Settings")
        int32 VertIndex = 0;
};

// ****
// Verlet - Class that represents a single vertex
// ****
class Verlet
{
public:
    Verlet()
    {
        ExternalForce = FVector(0.0f, 0.0f, 0.0f);
    }
};

```

```

        unsigned int Flags;
        FVector Pos;
        FVector PreviousPos;
        FVector ExternalForce;

        void AddForce(FVector Force)
        {
            ExternalForce += Force;
        }
    };

    class VerletConstraint
    {
public:
    Verlet* V1 = nullptr;
    Verlet* V2 = nullptr;
    float ConstraintDist = 0.0f;

    eVerletConstraintType Type = eVerletConstraintType::Normal;
};

// ****
// AVerletSim - Base class for all verlet simulation classes
// ****
UCLASS()
class FINALPROJECTRYANT_API AVerletSim : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AVerletSim();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick(float DeltaSeconds) override;

    float GetParticleMass() { return ParticleMass; }

    void CreateConstraint(Verlet* V1, Verlet* V2, eVerletConstraintType TypeIn)
    {
        VerletConstraint Constraint;

```

```

        Constraint.Type = TypeIn;

        Constraint.V1 = V1;
        Constraint.V2 = V2;
        Constraint.ConstraintDist = FVector::Dist( V1->Pos, V2->Pos );
        ConstraintList.Add(Constraint);
    }

protected:
    // Mass for each vertex of the rope
    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics", meta = (ClampMin = 0.0f))
        float ParticleMass = 5.0f;

    // The amount of damping to be imparted on the rope
    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics", meta = (ClampMin = 0.0f, DisplayName = "Damping Amount"))
        float DampingAmt = 0.01f;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
        int32 NumConstrainIterations = 15;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Collision")
        bool bEnableCollision = false;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Collision")
        bool bEnableStaticCollision = false;

    UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Physics")
        float WindScale = 1.0f;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
        bool bEnableSimulation = true;

    UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Physics")
        bool bEnableGravity = true;

    // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
    // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.

    // Toggles display of debug draw lines
    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings", meta = (DisplayName = "Enable Debug Draw"))
        bool bEnableDebugDraw = false;

    UPROPERTY(EditAnywhere, Category = "Verlet Settings|Debug Settings")

```

```

        bool bShowLockedVerts = false;  

UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Debug Settings")
    eShowForceMode ShowForcesMode = eShowForceMode::None;  

UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Debug Settings")
    eVerletConstraintType DrawConstraintFilter = eVerletConstraintType::All;  

UPROPERTY(EditAnywhere, Interp , Category = "Verlet Settings|Render Mode")
    eRenderMode RenderMode = eRenderMode::ProgressiveMesh;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Render Mode")
    bool bRenderModeIncremental = false;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Render Mode")
    float RenderModeIncrementalTime = 1.0f;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
    float EdgeLength = 10.0f;  

float RenderModeTimer = 0.0f;
int RenderModeIndex = 0;  

float Timer = 0.0f;                                // General timer for animation debug stuff  

AWindSim* Wind = nullptr;  

UPrimitiveComponent* LastAddedComponent = nullptr;  

UPROPERTY()
UProceduralMeshComponent* MeshComponent = nullptr;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
    UMaterial* MeshMaterial = nullptr;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Mesh")
    FVector2D UVTileScale = FVector2D(1.0f, 1.0f);  

TArray<VerletConstraint> ConstraintList;  

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Physics")
    TArray<FVerletAttachment> AttachmentList;  

UPROPERTY()

```

```

UTextContainerComponent* TextContainer = nullptr;

UPROPERTY(EditAnywhere, Interp, Category = "Verlet Settings|Text")
    float TextContainerIndex = -1; // -1 default = No Text

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Trigger")
    ULevelSequence* LevelSequence = nullptr;

UPROPERTY()
    ULevelSequencePlayer* SequencePlayer = nullptr;

UPROPERTY(EditAnywhere, Category = "Verlet Settings|Trigger")
    bool bEnableTriggerVolume = false;

UPROPERTY(EditAnywhere)
    UBoxComponent* TriggerVolume = nullptr;

UPROPERTY(EditAnywhere, Category = "Verlet Settings")
    float MaxTickRange = 500.0f;

protected:

void SimulateVert(Verlet& Vert, float DeltaTime);
bool ResolveVertCollision(Verlet& Vert);
void ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength);
void DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode, FColor Colour = FColor::Blue );
void UpdateSubTitleText();

void OnEndSequnceEvent();

UFUNCTION()
    void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
UFUNCTION()
    void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
OtherBodyIndex);

// Check if we are close enough to tick (Optimisation)
bool OutsideOfTickRange()
{
    if ( Timer < 10.0f ) // Must be allowed to Tick for at least 10 seconds
        false; // to allow everything to settle

FVector Pos = GetWorld()->GetFirstPlayerController()->GetPawn()->GetActorLocation();

```

```

        return FVector::Dist(Pos, GetActorLocation()) > MaxTickRange;
    }

};

```

### *VerletSim.cpp*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "DrawDebugHelpers.h"
#include "RyanTCode/Misc/MyStaticMesh.h"
#include "Kismet/GameplayStatics.h"
#include "FinalProjectRyanTHUD.h"
#include "VerletSim.h"

// *****
// Constructor
// *****
AVerletSim::AVerletSim()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));

    // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void AVerletSim::BeginPlay()
{
    Super::BeginPlay();

    // Finds the first wind source object in the level
    if (!Wind)
    {
        // GetAllActorsOfClass Searches the world and returns a list of all objects of the type required.
        // In this case we are looking for all AWindSim objects.
    }
}

```

```

TArray<AActor*> WindList;
UGameplayStatics::GetAllActorsOfClass(GetWorld(), AWindSim::StaticClass(), WindList);

if (WindList.Num())
    Wind = Cast<AWindSim>(WindList[0]); // Use the first one in the list
}

// Create the procedural mesh component
MeshComponent = NewObject<UProceduralMeshComponent>(this, NAME_None);
MeshComponent->RegisterComponent();
MeshComponent->SetWorldTransform(FTransform()); // Set transform to Position(0,0,0) and Rotation(0,0,0)

// KeepWorldTransform means that the component is not relative to the AVerletSim actor, it has it's own worldspace
position
MeshComponent->AttachToComponent(LastAddedComponent ? LastAddedComponent : GetRootComponent(),
FAttachmentTransformRules::KeepWorldTransform);
LastAddedComponent = MeshComponent;

// Get a list of all UTextContainerComponents
TArray<UActorComponent*> TextComponents = GetComponentsByClass(UTextContainerComponent::StaticClass());
if (TextComponents.Num() )
    TextContainer = Cast<UTextContainerComponent>(TextComponents[0]); // Use the first text container component
in list (Should only be one)

TextContainerIndex = -1;

TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);

if ( bEnableTriggerVolume )
{
    // Register the enter/exit volume callbacks
    TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AVerletSim::EnterVolume);
    TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AVerletSim::ExitVolume);
}
}

// ****
// Tick - Called every frame
// ****
void AVerletSim::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

```

```

Timer += DeltaTime;

// Update the visibility of the mesh. If in wire frame etc, we don't show the mesh
if ( RenderMode == eRenderMode::ProgressiveMesh || RenderMode == eRenderMode::ProgressiveMeshSquares || RenderMode == eRenderMode::ProgressiveMeshTriangles )
    MeshComponent->SetVisibility( true, true );
else
    MeshComponent->SetVisibility( false, true );

// Update the incremental wireframe timer and current index
if ( bRenderModeIncremental )
{
    RenderModeTimer += DeltaTime;
    if ( RenderModeTimer >= RenderModeIncrementalTime )
    {
        RenderModeTimer -= RenderModeIncrementalTime;
        RenderModeIndex++;
    }
}

// Set transform to Position(0,0,0) and Rotation(0,0,0) as the mesh verts are in world space, so no transform required
MeshComponent->SetWorldTransform(FTransform());

// Poll sequence player in it is playing to see if it has completed.
if (SequencePlayer && !SequencePlayer->IsPlaying())
    OnEndSequnceEvent();

UpdateSubTitleText();
}

// ****
// UpdateSubTitleText - Updates HUD text from TextContainerComponent using TextContainerIndex
// ****
void AVerletSim::UpdateSubTitleText()
{
    if (TextContainer && TextContainerIndex>= 0 && TextContainerIndex < TextContainer->GetNumText())
    {
        TextContainer->SetTextFromList(-1);                                // Turn off 3d text

        FString Temp = TextContainer->GetTextFromList(TextContainerIndex);

        // (Bug fix) Code to filter CR characters but leave LF to prevent engine showing double line spacing
        while ( true )
        {

```

```

        int32 Index;
        if ( !Temp.FindChar( TCHAR('\r'), Index ) )
            break;
        Temp.RemoveAt( Index, 1 );
    }
    AFinalProjectRyanTHUD::SetString( Temp );
}

// ****
// SimulateVert - Handles the simulation of verts
// ****
void AVerletSim::SimulateVert(Verlet& Vert, float DeltaTime)
{
    if ( !bEnableSimulation )
        return;

    FVector PreviousPos = Vert.Pos;

    // If disable simulation flag is set, return
    if (Vert.Flags)
    {
        if (bShowLockedVerts)
            DrawDebugSphere(GetWorld(), Vert.Pos, 10.0f, 8, FColor::White);
        return;
    }

    // Zero force to begin with
    FVector Force = Vert.ExternalForce;

    DrawForce( Vert.Pos, Vert.ExternalForce, ShowForcesMode, FColor::Blue );

    FVector NewForce;

    if ( bEnableGravity )
    {
        // Add gravity force
        NewForce = FVector(0.0f, 0.0f, -9.80f);           // TODO: Pass in UE4 gravity setting instead of hardcoding
        Force += NewForce;
        DrawForce( Vert.Pos, NewForce, ShowForcesMode, FColor::Turquoise );
    }

    // Add wind force
    if (Wind)

```

```

    {
        // Get the wind force for the world position of the current vertex.
        if (WindScale != 0.0f)
        {
            NewForce = Wind->GetWindForce(Vert.Pos) * WindScale;

            Force += NewForce;

            DrawForce( Vert.Pos, NewForce, ShowForcesMode, FColor::Red );
        }
    }

    // Dividing force by particle mass
    if (ParticleMass != 0.0f)           // Avoid divide by 0
        Force /= ParticleMass;

    Vert.Pos += ((Vert.Pos - Vert.PreviousPos) * DampingAmt) + (Force * DeltaTime);

    Vert.PreviousPos = PreviousPos;

    // Set force to zero after it has been applied to vert
    Vert.ExternalForce = FVector(0.0f, 0.0f, 0.0f);
}

// *****
// ResolveVertCollision - Handles Collision For Verts
// *****
bool AVerletSim::ResolveVertCollision(Verlet& Vert)
{
    if ( Vert.Flags )
        return false;                                // Attached vert, so don't perform collision.

    bool bCollided = false;

    //-----
    // Collision with static objects
    //-----
    if ( bEnableStaticCollision )
    {
        FHitResult HitResult;
        FCollisionQueryParams Params;
        FCollisionObjectQueryParams ObjParams;
        Params.AddIgnoredComponent( MeshComponent );      // Dont collide with self
    }
}

```

```

ObjParams.AddObjectTypeToQuery( ECollisionChannel::ECC_WorldStatic );

FVector RayStart = Vert.PreviousPos;
FVector RayEnd = Vert.Pos;

if (GetWorld()->LineTraceSingleByObjectType( HitResult, RayStart, RayEnd, ObjParams, Params))
{
    if ( HitResult.Actor.IsValid() && HitResult.Actor->GetClass() != AMyStaticMesh::StaticClass() )
    {
        float StandOff = 1.0f;                                // Collision stand off distance

        Vert.Pos           = HitResult.ImpactPoint + HitResult.ImpactNormal * StandOff;      // Move
out from impact a bit
        Vert.PreviousPos   = Vert.Pos;

        if ( HitResult.GetComponent() )
        {
            // If it's movable, add an impulse to it
            if ( HitResult.GetComponent()->IsSimulatingPhysics() && HitResult.GetComponent()->Mobility == EComponentMobility::Movable )
            {
                FVector Impulse = -HitResult.ImpactNormal * 40.0f;
                HitResult.GetComponent()->AddImpulseAtLocation( Impulse, HitResult.ImpactPoint );
            }
        }
        bCollided = true;
    }
}

//-----
// Collision with dynamic objects (Spheres only)
//-----
for ( int i = 0 ; i < AMyStaticMesh::MeshList.Num() ; i++ )
{
    AMyStaticMesh *MyMesh = AMyStaticMesh::MeshList[i];

    FVector SpherePos = MyMesh->GetActorLocation();

    float Dist = FVector::Dist( SpherePos, Vert.Pos );

    if ( Dist <= MyMesh->SphereRadius )
    {
        DrawDebugSphere(GetWorld(), Vert.Pos, 10.0f, 8, FColor::White);
    }
}

```

```

        FVector Dir = Vert.Pos - SpherePos;                                // Get vector from
sphere centre to the vert position                                         // Normalise it
        Dir.Normalize();

        FVector ImpactPoint = SpherePos + Dir * MyMesh->SphereRadius;      // Calculate impact point by moving out
from the centre of sphere towards the vert by the sphere radius.
        Vert.Pos           = ImpactPoint + Dir;
        Vert.PreviousPos   = Vert.Pos;

        bCollided = true;
    }
}
return bCollided;
}

// ****
// ConstrainVert - Constrains VertToConstrain to Vert
// ****
void AVerletSim::ConstrainVert(Verlet& Vert, Verlet& VertToConstrain, float ConstraintLength)
{
    if ( !bEnableSimulation )
        return;

    if (Vert.Flags && VertToConstrain.Flags)
        return;

    FVector& CurrentVert      = Vert.Pos;                                     // Local referece for clarity
    FVector NextVert           = VertToConstrain.Pos;                      // Local referece for clarity

    FVector Direction = NextVert - CurrentVert;                            // Direction vector from CurrentVert to NextVert

    float Distance = Direction.Size();                                      // Get the distance between them both

    Direction *= 1.0f - (ConstraintLength / Distance);                    // Get the percentage of change needed to move back into
correct position * Direction

    if (Vert.Flags || VertToConstrain.Flags)
    {
        // If it is an attached vert, constrain by the full percentage
        if (Vert.Flags)
            VertToConstrain.Pos -= Direction;
        else
            Vert.Pos += Direction;
    }
}

```

```

        }
    else
    {
        // If it is not, constrain both verts by half of the percentage in opposite directions so they constrain from the
line centre
        VertToConstrain.Pos -= Direction * 0.5f;
        Vert.Pos += Direction * 0.5f;
    }
}

// ****
// DrawForce - Draws a debug arrow to show force in world
// ****
void AVerletSim::DrawForce( const FVector &Pos, const FVector &Force, eShowForceMode Mode, FColor Colour )
{
    if ( Mode == eShowForceMode::None )
        return;

    float Magnitude = Force.Size();                                // Get the magnitue of the force (Speed)

    if ( Magnitude < 0.01f )                                       // If its too small to draw, just exit
        return;

    FVector Dir = Force / Magnitude;                               // Normalise dir by dividing by its magnitude

    float Length = 30.0f;                                         // Maximum length

    if ( Mode == eShowForceMode::ShowWithMagnitude )
    {
        // Scale Length my force magnitude
        Length = 10.0f * Magnitude;
    }
    else
    {
        if ( Mode == eShowForceMode::ShowWithModulation )
        {
            // Apply some modulation to the length if needed

            float Modulation = sinf(Timer*2.0f);                  // Get modulating value from -1 to +1
            Modulation += 1.0f;                                     // Add 1 to put in range 0 to 2
            Modulation *= 0.5f;                                    // divide by 2 to put in range 0 to 1
            Length *= Modulation;                                 // Scale Length by the modulating value from 0.0
to 1.0
        }
    }
}

```

```

    }

    Dir *= Length;                                     // Multiple Dir by Length to get offset to the
end of the arrow

    DrawDebugDirectionalArrow( GetWorld(), Pos, Pos + Dir, 7.0f, Colour, false, 0.0f, 0, 1.0f);
}

// *****
// EnterVolume - Called when component enters collision volume
// *****
void AVerletSim::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Start the level sequence if we have one
    if ( LevelSequence )
    {
        FLevelSequencePlaybackSettings Settings;
        Settings.LoopCount = 0;
        SequencePlayer = ULevelSequencePlayer::CreateLevelSequencePlayer(GetWorld(), LevelSequence, Settings);
        SequencePlayer->Play();
    }
}

// *****
// OnEndSequnceEvent - Triggered when level sequence has finished playing
// *****
void AVerletSim::OnEndSequnceEvent()
{
    // Stop the level sequence
    if ( SequencePlayer )
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }
    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();
}

```

```

// ****
// EnterVolume - Called when component exits collision volume
// ****
void AVerletSim::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Stop the level sequence
    if ( SequencePlayer )
    {
        SequencePlayer->Stop();
        SequencePlayer = nullptr;
    }

    // Reset the HUD SubTitle text
    TextContainerIndex = -1;
    AFinalProjectRyanTHUD::SetString();
}

```

### *ClothSim.h*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "VerletSim.h"
#include "Components/BoxComponent.h"
#include "ClothSim.generated.h"

UCLASS()
class FINALPROJECTRYANT_API AClothSim : public AVerletSim
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AClothSim();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

```

```

// Called every frame
virtual void Tick(float DeltaSeconds) override;

// The number of horizontal verts in the cloth
UPROPERTY(EditAnywhere, Category = "Verlet Settings")
int NumberOfVertsW = 35;

// The number of vertical verts in the cloth
UPROPERTY(EditAnywhere, Category = "Verlet Settings")
int NumberOfVertsH = 20;

private:
    void Constrain();
    void Simulate(float DeltaTime);
    void Render();
    void ResolveCollision();

    void DebugDraw(int y, int z);

    void DrawWireframeTriangles( bool bIncremental );
    void DrawWireframeSquares( bool bIncremental );
    void DrawWireframeConstraints( bool bIncremental );

    void BuildMesh();
    void BuildConstraints();
    void UpdateMesh();
    void UpdateLockedVerts();

private:
    // The maximum that NumberOfVertsW and NumberOfVertsH can be
    static const int MaxVertsW = 50;
    static const int MaxVertsH = 50;

    // 2 dimensional array of verts in the cloth
    Verlet Verts[MaxVertsW][MaxVertsH];
};

ClothSim.cpp
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"

```

```

#include "Engine.h"
#include "ClothSim.h"
#include "DrawDebugHelpers.h"
#include "Kismet/GameplayStatics.h"
#include "SceneManagement.h"
#include "FinalProjectRyanTHUD.h"

// For UE4 Profiler: Type stat Ryan in console to see timing stats
DECLARE_CYCLE_STAT(TEXT("Cloth Constrain") , STAT_ClothConstrain , STATGROUP_Ryan); // Stat
within group 'Ryan' See "FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Cloth UpdateMesh") , STAT_ClothUpdateMesh , STATGROUP_Ryan); // Stat
within group 'Ryan' See "FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Cloth ResolveCollision") , STAT_ClothResolveCollision , STATGROUP_Ryan); // Stat within group
'Ryan' See "FinalProjectRyanT.h"
DECLARE_CYCLE_STAT(TEXT("Cloth Simulate") , STAT_ClothSimulate , STATGROUP_Ryan); // Stat
within group 'Ryan' See "FinalProjectRyanT.h"

// *****
// Constructor
// *****
AClothSim::AClothSim()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void AClothSim::BeginPlay()
{
    Super::BeginPlay();

    FTransform Transform = GetActorTransform(); // Get the transform of this actor
    FVector SideAxis = Transform.GetUnitAxis(EAxis::Y); // Get the vector that points to the side
    FVector UpAxis = Transform.GetUnitAxis(EAxis::Z); // Get the vector that points up

    // Initialising all verts in cloth
    FVector Pos = Transform.GetLocation();
    for (int z = 0; z < NumberOfVertsH; z++)
    {
        FVector OldPos = Pos; // Remember position of left side of cloth

        for (int y = 0; y < NumberOfVertsW; y++)

```

```

    {
        Verts[y][z].Flags = 0;                                // Reset the flags
        Verts[y][z].Pos = Pos;                                // Set the position
        Verts[y][z].PreviousPos = Pos;                         // Set the previous position to the same
        Pos += SideAxis * EdgeLength;                          // Move across the SideAxis by ConstrainDist
    world units
    }
    Pos = OldPos;                                         // Resets position to left side
    Pos -= UpAxis * EdgeLength;                           // Move down the UpAxis by ConstrainDist
    world units
}

BuildConstraints();                                     // Build the constraint list
BuildMesh();                                           // Construct the mesh using the above
initialised verts
}

// *****
// Tick - Called every frame
// *****
void AClothSim::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Check if we are close enough to tick (Optimisation)
    if (OutsideOfTickRange())
        return;

    UpdateLockedVerts();                                  // Update any verts that are locked
    Simulate(DeltaTime);                               // Tick the simulation for all vert

    // Constraint loop to iron out any inconsistencies caused by collision/constrain 'pushing' and 'pulling' the verts
    for (int i = 0; i < NumConstrainIterations; i++)
    {
        Constrain();                                 // Constrain()
        ResolveCollision();                         // ResolveCollision()
    }

    Render();                                            // Render debug stuff / update the
mesh
}

// *****

```

```

// ResolveCollision - Resolve collisions for all verts (Based on the velocity)
// ****
void AClothSim::ResolveCollision()
{
    if ( !bEnableCollision )
        return;

    SCOPE_CYCLE_COUNTER(STAT_ClothResolveCollision); // For UE4 profiling

    for (int z = 0; z < NumberOfVertsH; z++)
    {
        for (int y = 0; y < NumberOfVertsW; y++)
        {
            // Resolve collision for current vert
            if (ResolveVertCollision(Verts[y][z]))
            {
                // Listener for rope debug draw tickbox
                if (bEnableDebugDraw)
                    DebugDraw(y, z);
            }
        }
    }
}

// ****
// Constrain - Maintain equal distance between all verts
// ****
void AClothSim::Constrain()
{
    SCOPE_CYCLE_COUNTER(STAT_ClothConstrain); // For UE4 profiling

    for (int i = 0; i < ConstraintList.Num() ; i++)
    {
        ConstrainVert(*ConstraintList[i].V1, *ConstraintList[i].V2, ConstraintList[i].ConstraintDist);
    }
}

// ****
// Simulate - Simulate verlet physics for rope
// ****
void AClothSim::Simulate(float DeltaTime)
{
    SCOPE_CYCLE_COUNTER(STAT_ClothSimulate); // For UE4 profiling
}

```

```

        for (int z = 0; z < NumberOfVertsH; z++)
            for (int y = 0; y < NumberOfVertsW; y++)
                SimulateVert(Verts[y][z], DeltaTime);
    }

// ****
// DrawWireframeTriangles - Draws the mesh as wireframe triangles
// ****
void AClothSim::DrawWireframeTriangles( bool bIncremental )
{
    if ( bRenderModeIncremental )
    {
        int NumTriangles = (NumberOfVertsW-1)*(NumberOfVertsH-1) * 2;

        if ( RenderModeIndex >= NumTriangles )
            RenderModeIndex = 0;
    }

    FColor Colour = FColor::Green;

    int Count = 0;
    for (int z = 0; z < NumberOfVertsH - 1; z++)
    {
        for (int y = 0; y < NumberOfVertsW - 1; y++)
        {
            //A-0,0----0,1-B
            //   | \   |
            //   | \   |
            //   | \   |
            //   | \   |
            //C-0,1----1,1-D

            FVector& A = Verts[y][z].Pos;
            FVector& B = Verts[y][z+1].Pos;
            FVector& C = Verts[y+1][z].Pos;
            FVector& D = Verts[y+1][z+1].Pos;

            if ( bRenderModeIncremental && Count > RenderModeIndex )
                break;

            // Triangle 1
            DrawDebugLine(GetWorld(), A, B, Colour, false, -1.0f, 0, 1.0f);
            DrawDebugLine(GetWorld(), B, D, Colour, false, -1.0f, 0, 1.0f);
            DrawDebugLine(GetWorld(), D, A, Colour, false, -1.0f, 0, 1.0f);
        }
    }
}

```

```

        Count++;

        if ( bRenderModeIncremental && Count > RenderModeIndex )
            break;

        // Triangle 2
        DrawDebugLine( GetWorld(), A, D, Colour, false, -1.0f, 0, 1.0f );
        DrawDebugLine( GetWorld(), D, C, Colour, false, -1.0f, 0, 1.0f );
        DrawDebugLine( GetWorld(), C, A, Colour, false, -1.0f, 0, 1.0f );
        Count++;
    }
}

// *****
// DrawWireframeSquares - Draws the mesh as wireframe squares
// *****
void AClothSim::DrawWireframeSquares( bool bIncremental )
{
    if ( bRenderModeIncremental )
    {
        if ( RenderModeIndex >= ((NumberOfVertsW-1)*(NumberOfVertsH-1)) )
            RenderModeIndex = 0;
    }

    // For each vert, draw a debug line, alternating between the colours red and blue.
    int Count = 0;
    for (int z = 0; z < NumberOfVertsH - 1; z++)
    {
        bool bIsLastRow = z == (NumberOfVertsH - 2);

        for (int y = 0; y < NumberOfVertsW - 1; y++)
        {
            if ( bRenderModeIncremental && Count > RenderModeIndex )
                break;

            bool bIsLastColumn = y == (NumberOfVertsW - 2);

            //A-0,0----1,0-B
            //  | \  |
            //  | \  |
            //  | \  |
            //C-0,1----1,1-D

```

```

FVector& A = Verts[y][z].Pos;
FVector& B = Verts[y+1][z].Pos;
FVector& C = Verts[y][z+1].Pos;
FVector& D = Verts[y+1][z+1].Pos;

FColor Colour = FColor::Green;

// Draw top line
DrawDebugLine(GetWorld(), A, B, Colour, false, -1.0f, 0, 1.0f);

if ( bRenderModeIncremental || bIsLastRow )
{
    // Draw Bottom line if we are drawing the last row
    DrawDebugLine(GetWorld(), C, D, Colour, false, -1.0f, 0, 1.0f);
}

// 
Colour = FColor::Red;

// Draw left line
DrawDebugLine(GetWorld(), A, C, Colour, false, -1.0f, 0, 1.0f);

if ( bRenderModeIncremental || bIsLastColumn )
{
    // Draw Right line if we are drawing the last column
    DrawDebugLine(GetWorld(), B, D, Colour, false, -1.0f, 0, 1.0f);
}
Count++;
}

}

}

// *****
// DrawWireframeConstraints - Draws all constraints
// *****
void AClothSim::DrawWireframeConstraints( bool bIncremental )
{
    for (int i = 0; i < ConstraintList.Num() ; i++)
    {
        // Don't show diagonal constraints
        if ( DrawConstraintFilter != eVerletConstraintType::All && ConstraintList[i].Type != DrawConstraintFilter )
            continue;

        FColor Colour = FColor::Green;

```

```

float Length = (ConstraintList[i].V1->Pos - ConstraintList[i].V2->Pos).Size();

// Apply some modulation to the length

float Modulation = sinf(Timer*5.0f); // Get modulating value from -1 to +1
Modulation += 1.0f; // Add 1 to put in range 0 to 2
Modulation *= 0.5f; // divide by 2 to put in range 0 to 1

Modulation *= 0.3f; // Change to range 0.7 to 1.0
Modulation += 0.7f;

Length *= Modulation; // Scale Length by the modulating value from 0.0 to 1.0

#if 1
    // 2 arrows, outwards from line centre
    FVector Centre = (ConstraintList[i].V1->Pos + ConstraintList[i].V2->Pos) * 0.5f; // Centre of the line
= the average of both positions

    FVector Dir = (ConstraintList[i].V1->Pos - ConstraintList[i].V2->Pos);
    Dir.Normalize();

    FVector Offset = Dir * Length * 0.5f;

    DrawDebugDirectionalArrow( GetWorld(), Centre, Centre + Offset, 7.0f, FColor::Magenta, false, 0.0f, 0, 1.0f);
    DrawDebugDirectionalArrow( GetWorld(), Centre, Centre - Offset, 7.0f, FColor::Magenta, false, 0.0f, 0, 1.0f);

#else
    // 1 arrow from vert 1 to vert 2
    FVector Dir = (ConstraintList[i].V2->Pos - ConstraintList[i].V1->Pos);
    Dir.Normalize();

    FVector Offset = Dir * Length;

    DrawDebugDirectionalArrow( GetWorld(), ConstraintList[i].V1->Pos, ConstraintList[i].V1->Pos + Offset, 7.0f,
FColor::Magenta, false, 0.0f, 0, 1.0f);
#endif
}

// ****
// Render - Render debug stuff / update the mesh
// ****
void AClothSim::Render()
{

```

```

switch ( RenderMode )
{
    case eRenderMode::None:
        break;

    case eRenderMode::WireframeTriangles:
    {
        DrawWireframeTriangles( bRenderModeIncremental );
        break;
    }

    case eRenderMode::WireframeSquares:
    {
        DrawWireframeSquares( bRenderModeIncremental );
        break;
    }

    case eRenderMode::WireframeConstraints:
    {
        DrawWireframeConstraints( bRenderModeIncremental );
        break;
    }

    case eRenderMode::WireframeSquaresAndConstraints:
    {
        DrawWireframeSquares( false /*bRenderModeIncremental*/ );
        DrawWireframeConstraints( bRenderModeIncremental );
        break;
    }

    case eRenderMode::ProgressiveMesh:
    {
        UpdateMesh();
        break;
    }

    case eRenderMode::ProgressiveMeshSquares:
    {
        UpdateMesh();
        DrawWireframeSquares( bRenderModeIncremental );
        break;
    }

    case eRenderMode::ProgressiveMeshTriangles:

```

```

        {
            UpdateMesh();
            DrawWireframeTriangles( bRenderModeIncremental );
            break;
        }
    }

// ****
// DebugDraw - Draws a sphere at a given verts position
// ****
void AClothSim::DebugDraw(int y, int z)
{
    DrawDebugSphere(GetWorld(), Verts[y][z].Pos, 10.0f, 8, FColor::White);
}

// ****
// BuildMesh - Build the textured mesh
// ****
void AClothSim::BuildMesh()
{
    MeshComponent->ClearAllMeshSections();           // Clear any old meshes

    TArray <FVector>             Vertices;
    TArray <int32>                Triangles;
    TArray <FVector>             Normals;
    TArray <FVector2D>            UV0;
    TArray <FColor>               VertexColor;
    TArray <FPProcMeshTangent>   VertexCTangent;

    // Build the vertex list
    for (int z = 0; z < NumberOfVertsH; z++)
    {
        for (int y = 0; y < NumberOfVertsW; y++)
        {
            Vertices.Add( Verts[y][z].Pos );

            Normals.Add( FVector(1,0,0) );

            FVector2D UV = FVector2D(y,z);
            UV.X /= float(NumberOfVertsW);
            UV.Y /= float(NumberOfVertsH);

            UV *= UVTileScale;
        }
    }
}

```

```

        UV0.Add(UV);
    }

    // Build the triangle list. Each trianlge has 3 x Indices into the above vertex list.
    //
    //      0----1
    //      | \   |
    //      |  \  |
    //      |   \ |
    //      |     \| 
    //      W+0--W+1
    //
    for (int z = 0; z < NumberOfVertsH-1; z++)
    {
        for (int y = 0; y < NumberOfVertsW-1; y++)
        {
            int32 BaseIndex = y+(z*NumberOfVertsW);

            Triangles.Add( BaseIndex + 0 );
            Triangles.Add( BaseIndex + NumberOfVertsW );
            Triangles.Add( BaseIndex + NumberOfVertsW + 1 );

            Triangles.Add( BaseIndex + NumberOfVertsW+1 );
            Triangles.Add( BaseIndex + 1 );
            Triangles.Add( BaseIndex + 0 );
        }
    }

    bool bCreateCollision = true;
    MeshComponent->CreateMeshSection( 0, Vertices, Triangles, Normals, UV0, VertexColor, VertexCTangent, bCreateCollision );

    MeshComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_Vehicle, ECollisionResponse::ECR_Block);
    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldStatic, ECollisionResponse::ECR_Block);
    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldDynamic, ECollisionResponse::ECR_Block);
    MeshComponent->SetCollisionResponseToChannel(ECollisionChannel::ECC_PhysicsBody, ECollisionResponse::ECR_Block);
    MeshComponent->SetNotifyRigidBodyCollision(true);

    MeshComponent->SetMaterial(0,MeshMaterial);
}

// ****

```

```

// UpdateMesh - Update the textured mesh
// ****
void AClothSim::UpdateMesh()
{
    // Exit if the render mode is set to not needing the mesh (wireframe etc)
    if ( RenderMode != eRenderMode::ProgressiveMesh && RenderMode != eRenderMode::ProgressiveMeshSquares && RenderMode != eRenderMode::ProgressiveMeshTriangles )
        return;

    SCOPE_CYCLE_COUNTER(STAT_ClothUpdateMesh);           // For UE4 profiling

    // Lists to contain all elements for the verts to pass to engine call
    TArray<FVector>             Vertices;
    TArray<FVector>             Normals;
    TArray<FVector2D>            UV0;
    TArray<FCOLOR>               VertexColor;
    TArray<FPProcMeshTangent>   VertexCTangent;

    // List for surface normals
    FVector SurfaceNormals[MaxVertsW][MaxVertsH];

    for (int z = 0; z < NumberOfVertsH; z++)
    {
        for (int y = 0; y < NumberOfVertsW; y++)
        {
            // Add the vert to the list
            Vertices.Add(Verts[y][z].Pos);

            // Create a surface normal ready for next stage (Vertex Normal)
            FVector Normal, V1, V2;
            if (y < (NumberOfVertsW-1) )
                V1 = Verts[y+1][z].Pos - Verts[y][z].Pos;
            else
                V1 = Verts[y][z].Pos - Verts[y-1][z].Pos;

            if (z < (NumberOfVertsH-1) )
                V2 = Verts[y][z+1].Pos - Verts[y][z].Pos;
            else
                V2 = Verts[y][z].Pos - Verts[y][z-1].Pos;

            // Add the surface normal to a local list ready for the next stage
            SurfaceNormals[y][z] = FVector::CrossProduct( V1, V2 );
            SurfaceNormals[y][z].Normalize();
        }
    }
}

```

```

}

// Build the vertex normals from the surface normals
for (int z = 0; z < NumberOfVertsH; z++)
{
    for (int y = 0; y < NumberOfVertsW; y++)
    {
        FVector Normal(0.0f,0.0f,0.0f);

        // Get the average of all neighbouring face normals to get the vertex normal at this vert
        if ( z > 0 )
            Normal += SurfaceNormals[y][z-1];
        if ( z < (NumberOfVertsH-1) )
            Normal += SurfaceNormals[y][z+1];

        if ( y > 0 )
            Normal += SurfaceNormals[y-1][z];
        if ( y < (NumberOfVertsW-1) )
            Normal += SurfaceNormals[y+1][z];

        Normal.Normalize();
        Normals.Add( Normal );
    }
}

// Call engine function to update the mesh
MeshComponent->UpdateMeshSection( 0, Vertices, Normals, UV0, VertexColor, VertexCTangent );
}

// ****
// BuildConstraints - Build constraints for all verts
// ****
void AClothSim::BuildConstraints()
{
    // Connect neighbouring verts with constraints (right, down, diagonal right, Diagonon left).
    for (int x = 0; x<NumberOfVertsW; x++)
    {
        for (int y = 0; y<NumberOfVertsH; y++)
        {
            if (x<NumberOfVertsW - 1)
                CreateConstraint(&Verts[x][y], &Verts[x + 1][y], eVerletConstraintType::Normal);

            if (y<NumberOfVertsH - 1)
                CreateConstraint(&Verts[x][y], &Verts[x][y + 1], eVerletConstraintType::Normal);
        }
    }
}

```

```

        if (x<NumberOfVertsW - 1 && y<NumberOfVertsH - 1)
    {
        CreateConstraint(&Verts[x][y], &Verts[x + 1][y + 1], eVerletConstraintType::Diagonal);
        CreateConstraint(&Verts[x + 1][y], &Verts[x][y + 1], eVerletConstraintType::Diagonal);
    }
}

// Connect neighbouring verts with secondary constraints (right, down, diagonal right, Diagonon left).
// These span 2 cells, so they also constrain the curvature of the cloth surface too
for (int x = 0; x<NumberOfVertsW; x++)
{
    for (int y = 0; y<NumberOfVertsH; y++)
    {
        if (x<NumberOfVertsW - 2)
            CreateConstraint(&Verts[x][y], &Verts[x + 2][y], eVerletConstraintType::NormalSeconary);

        if (y<NumberOfVertsH - 2)
            CreateConstraint(&Verts[x][y], &Verts[x][y + 2], eVerletConstraintType::NormalSeconary);

        if (x<NumberOfVertsW - 2 && y<NumberOfVertsH - 2)
        {
            CreateConstraint(&Verts[x][y], &Verts[x + 2][y + 2], eVerletConstraintType::DiagonalSeconary);
            CreateConstraint(&Verts[x + 2][y], &Verts[x][y + 2], eVerletConstraintType::DiagonalSeconary);
        }
    }
}

// ****
// UpdateLockedVerts - Update any verts that are locked
// ****
void AClothSim::UpdateLockedVerts()
{
    // Clear all flags (Allows real-time editing of attach points)
    for (int z = 0; z < NumberOfVertsH; z++)
        for (int y = 0; y < NumberOfVertsW; y++)
            Verts[y][z].Flags = 0;

    FTransform Transform = GetActorTransform();

    for ( int i = 0 ; i < AttachmentList.Num() ; i++ )
    {

```

```

int Index = AttachmentList[i].VertIndex;

if ( Index >= 0 && Index < NumberOfVertsW*NumberOfVertsH )
{
    int Horiz = Index % NumberOfVertsW;                                // Get horizontal index from vertex index
    int Verti = Index / NumberOfVertsW;                                  // Get vertical index from vertex index

    FVector Pos = AttachmentList[i].AttachPos;
    if ( AttachmentList[i].bRelativeToActor )
        Pos = Transform.TransformPosition(Pos);                         // Transform into World Space as Attachment List
positions are in local space

        // Force the vert to the position of the attachment point
    Verts[Horiz][Verti].Pos = Pos;
    Verts[Horiz][Verti].PreviousPos = Pos;
    Verts[Horiz][Verti].Flags = 1;                                       // Tell the simulation not to simulate for this
vert or move during constraining
}
}
}

```

## Projectiles

```

StickProjectile.h
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "FinalProjectRyanTProjectile.h"
#include "StickProjectile.generated.h"

/**
 *
 */
UCLASS()
class FINALPROJECTRYANT_API AStickProjectile : public AFinalProjectRyanTProjectile
{
GENERATED_BODY()

/** called when projectile hits something */
UFUNCTION()

```

```
    virtual void OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector  
NormalImpulse, const FHitResult& Hit) override;  
};
```

### *StickProjectile.cpp*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```
#include "FinalProjectRyanT.h"  
#include "StickProjectile.h"  
#include "DrawDebugHelpers.h"  
  
// ****  
// OnHit - Callback for when projectile hits another actor or component  
// ****  
void AStickProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector  
NormalImpulse, const FHitResult& Hit)  
{  
    // Only add impulse and destroy projectile if we hit a physics  
    if ((OtherActor != NULL) && (OtherActor != this) && (OtherComp != NULL))  
    {  
        // Attach the projectile to whatever it hits  
        AttachToActor(OtherActor, FAttachmentTransformRules::KeepWorldTransform);  
  
        // Add a physics impulse to the other actor to make it react  
        if (OtherComp->IsSimulatingPhysics())  
            OtherComp->AddImpulseAtLocation(GetVelocity() * 100.0f, GetActorLocation());  
  
        UE_LOG(LogTemp, Warning, TEXT("hit %s"), *OtherComp->GetName());  
        // Destroy();  
    }  
}
```

### *ProjectileVolume.h*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```
#pragma once  
  
#include "GameFramework/Actor.h"  
#include "ProjectileVolume.generated.h"  
  
UCLASS()  
class FINALPROJECTRYANT_API AProjectileVolume : public AActor
```

```

{
GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AProjectileVolume();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    // Trigger Volume: When entered the left/right projectile type will be setup for player
    UPROPERTY(EditAnywhere)
        UBoxComponent* TriggerVolume = nullptr;

    // The projectile type that is mapped to the left mouse button for this volume
    UPROPERTY(EditAnywhere, Category = "Projectiles")
        eProjectileType LeftButtonType = eProjectileType::None;

    // The projectile type that is mapped to the right mouse button for this volume
    UPROPERTY(EditAnywhere, Category = "Projectiles")
        eProjectileType RightButtonType = eProjectileType::None;

    // For remembering setting on entry
    eProjectileType OldLeft;
    eProjectileType OldRight;

    // Volume enter/exit callbacks
    UFUNCTION()
        void EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
    UFUNCTION()
        void ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32
OtherBodyIndex);
};


```

### *ProjectileVolume.cpp*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```
#include "FinalProjectRyanT.h"
#include "FinalProjectRyanTHUD.h"
```

```

#include "FinalProjectRyanTCharacter.h"
#include "ProjectileVolume.h"

// *****
// Constructor
// *****
AProjectileVolume::AProjectileVolume()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // UBoxComponent is used to create a trigger volume around the actor for triggering a demo level sequence etc.
    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxTrigger"));

    RootComponent = TriggerVolume;
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
void AProjectileVolume::BeginPlay()
{
    Super::BeginPlay();

    TriggerVolume->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
    TriggerVolume->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);

    TriggerVolume->OnComponentBeginOverlap.AddDynamic(this, &AProjectileVolume::EnterVolume);
    TriggerVolume->OnComponentEndOverlap.AddDynamic(this, &AProjectileVolume::ExitVolume);
}

// *****
// Tick - Called every frame
// *****
void AProjectileVolume::Tick( float DeltaTime )
{
    Super::Tick( DeltaTime );
}

// *****
// EnterVolume - Called when component enters collision volume
// *****
void AProjectileVolume::EnterVolume(UPrimitiveComponent* OverlappedComp, AActor* Other, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)

```

```

{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Get a pointer to the player character
    AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);

    if ( !Player )
        return;

    // Remember Old Setting
    OldLeft      = Player->ProjectileClassIndex;
    OldRight     = Player->ProjectileClassIndexRight;

    // Set players projectile types from this volume
    Player->ProjectileClassIndex      = LeftButtonType;
    Player->ProjectileClassIndexRight = RightButtonType;
}

// *****
// EnterVolume - Called when component exits collision volume
// *****
void AProjectileVolume::ExitVolume(UPrimitiveComponent* OtherComp, AActor* Other, UPrimitiveComponent* OverlappedComp, int32 OtherBodyIndex)
{
    // Only allowed if overlapped by any kind of character
    if ( !Other->GetClass()->IsChildOf( ACharacter::StaticClass() ) )
        return;

    // Get a pointer to the player character
    AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>(Other);

    if ( !Player )
        return;

    // Restore old settings
    Player->ProjectileClassIndex      = OldLeft;
    Player->ProjectileClassIndexRight = OldRight;
}

```

## Wind

*WindSim.h*

```
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#pragma once

#include "GameFramework/Actor.h"
#include "WindSim.generated.h"

UCLASS()
class FINALPROJECTTRYANT_API AWindSim : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AWindSim();

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called every frame
    virtual void Tick(float DeltaSeconds) override;

    FVector GetWindForce(FVector WorldPosition);

    // UPROPERTY exposes a variable to the Unreal Editor. This allow it to be changed at runtime in the details tab.
    // You can set various parameters here to affect how it is shown in the editor, such as display name, and min/max values.

    // Strength value of the wind
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (ClampMin = 0.0f))
        float Strength = 3.0f;

    // Toggles display of debug draw lines
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (DisplayName = "Enable Debug Draw"))
        bool bEnableDebugDraw = false;

    // Value for frequency of gusts
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (ClampMin = 0.0f))
        float GustFrequency = 1.0f;

    // Wavelength of gusts on the X axis
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (ClampMin = 0.0f))
```

```

        float GustXWavelength = 0.06f;

    // Wavelength of gusts on the Y axis
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (ClampMin = 0.0f))
        float GustYWavelength = 0.03f;

    // Wavelength of gusts on the Z axis
    UPROPERTY(EditAnywhere, Category = "Wind Settings", meta = (ClampMin = 0.0f))
        float GustZWavelength = 0.06f;

    // Time, in seconds, since the wind object was created. Used in GetWindForce to pass into sinf function to get continuous
    sine modulation.
    float Time = 0.0f;

private:
    FVector GetWindMultiplierAtLocation(FVector WorldPos);

    void DebugDraw();

};


```

### *WindSim.cpp*

```

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "RyanTCode/Wind/WindSim.h"
#include "DrawDebugHelpers.h"

// *****
// Constructor
// *****
AWindSim::AWindSim()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // A USceneComponent is required to transform, rotate and scale an actor in worldspace.
    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("SceneComponent"));
}

// *****
// BeginPlay - Called when the game starts or when spawned
// *****
```

```

void AWindSim::BeginPlay()
{
    Super::BeginPlay();
}

// ****
// Tick - Called every frame
// ****
void AWindSim::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    Time += DeltaTime;

    // Listener for debug draw tickbox
    if (bEnableDebugDraw)
        DebugDraw();
}

// ****
// GetWindForce - Returns the force of the wind
// ****
FVector AWindSim::GetWindForce(FVector WorldPosition)
{
    FVector WindPos = GetActorLocation();                                // Get the position of the wind. Used to calculate wind direction.

    FVector Force = WorldPosition - WindPos;                            // Get the vector pointing from the WindPos towards the WorldPosition

    Force.Normalize();                                                 // Normalises the force vector, so it is only one world

unit long

    FVector WindMultiplier = GetWindMultiplierAtLocation(WorldPosition); // Get an x,y,z scale value for forces based of

current wind params.

    return Force * WindMultiplier * Strength;                           // Returns the Force multiplied by WindMultiplier multiplied by

Strength
}

// ****
// GetWindMultiplierAtLocation - Get an x,y,z scale value for forces based of current wind params
// ****
FVector AWindSim::GetWindMultiplierAtLocation(FVector WorldPos)
{
    // Values will be different depending on the world position passed in so that the wind varies based of position

```

```

float T = Time * GustFrequency;                                     // Phase multiplier (Speed)
float ModX = WorldPos.X * GustXWavelength;                      // Wavelength X
float ModY = WorldPos.Y * GustYWavelength;                      // Wavelength Y
float ModZ = WorldPos.Z * GustZWavelength;                      // Wavelength Z

ModX = (1.0f + sinf(ModX + T))*0.5f;                            // Get sine value (-1 to +1) and convert into range 0 to 1
ModY = (1.0f + sinf(ModY + T))*0.5f;
ModZ = (1.0f + sinf(ModZ + T))*0.5f;

return FVector(ModX, ModY, ModZ);                                  // Return values are from 0.0 to 1.0 and are used to scale the wind
direction
}

// *****
// DebugDraw - Debug display for wind
// *****
void AWindSim::DebugDraw()
{
    FVector Centre = GetActorLocation();

    for (float X = -200; X < 200; X += 40)
    {
        for (float Y = -200; Y < 200; Y += 40)
        {
            for (float Z = -200; Z < 200; Z += 40)
            {
                FVector Pos = Centre + FVector(X, Y, Z);

                FVector Force = GetWindForce(Pos);

                DrawDebugLine(GetWorld(), Pos, Pos + Force, FColor::Green, false, -1.0f, 0, 3.0f);
            }
        }
    }
}

```

Modified Default Project Code

*FinalProjectRyanTHUD.h*

// Copyright © 2018 Ryan Tinman, All Rights Reserved.

```
#pragma once
#include "GameFramework/HUD.h"
```

```

#include "FinalProjectRyanTHUD.generated.h"

UCLASS()
class AFinalProjectRyanTHUD : public AHUD
{
    GENERATED_BODY()

public:
    AFinalProjectRyanTHUD();

    virtual void DrawHUD() override; // Primary draw call for the HUD

    static void SetString( FString String = "" ) // Setter to set/clear sub-title text
    {
        HUDText = String;
    }

private:
    bool DrawSubTitleText(); // Displays description text at bottom of screen in black box
    bool DrawProjectileText(); // Displays description text to show projectiles that are
equipped

private:
    class UTexture2D* CrosshairTex; // Crosshair asset pointer

    static FString HUDText; // Static text for sub-title box

    UFont* Font = nullptr; // Font used to display sub-title text
};

FinalProjectRyanTHUD.cpp
// Copyright © 2018 Ryan Tinman, All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "FinalProjectRyanTHUD.h"
#include "Engine/Canvas.h"
#include "TextureResource.h"
#include "CanvasItem.h"
#include "Engine.h"
#include "FinalProjectRyanTCharacter.h"

FString AFinalProjectRyanTHUD::HUDText = ""; // Static text for sub-title box

// ****

```

```

// Constructor
// ****
AFinalProjectRyanTHUD::AFinalProjectRyanTHUD()
{
    // Set the crosshair texture
    static ConstructorHelpers::FObjectFinder<UTexture2D>
CrosshairTexObj(TEXT("/Game/FirstPerson/Textures/FirstPersonCrosshair"));
    CrosshairTex = CrosshairTexObj.Object;

    // Load the font used to display sub-title text
//    ConstructorHelpers::FObjectFinder<UFont> FontObject(TEXT("Font'/Game/FinalProjectRyanT/Fonts/ARIAL_18'"));
//    ConstructorHelpers::FObjectFinder<UFont> FontObject(TEXT("Font'/Game/FinalProjectRyanT/Fonts/ARIAL_28'"));
    if (FontObject.Object)
        Font = FontObject.Object;

    SetString();           // Empty the text string
}

// ****
// DrawHUD - Draw whole HUD
// ****
void AFinalProjectRyanTHUD::DrawHUD()
{
    Super::DrawHUD();

    // Draw very simple crosshair

    // find center of the Canvas
    const FVector2D Center(Canvas->ClipX * 0.5f, Canvas->ClipY * 0.5f);

    // offset by half the texture's dimensions so that the center of the texture aligns with the center of the Canvas
    const FVector2D CrosshairDrawPosition( (Center.X),
                                         (Center.Y + 20.0f));

    // Draw the sub-title box
    if ( !DrawSubTitleText() )
    {
        // Draw the crosshair
        FCanvasTileItem TileItem( CrosshairDrawPosition, CrosshairTex->Resource, FLinearColor::White );
        TileItem.BlendMode = SE_BLEND_Translucent;
        Canvas->DrawItem( TileItem );

        // Draw text for equipped Projectiles
        DrawProjectileText();
    }
}

```

```

        }

// *****
// DrawSubTitleText - Displays description text at bottom of screen in black box
// *****
bool AFinalProjectRyanTHUD::DrawSubTitleText()
{
    if ( !Font || HUDText.IsEmpty() )           // If font not loaded or empty text, return
        return false;

    // Width & Height of the box
    float BGWidth   = Canvas->ClipX;           // ClipX = Full screen width
    float BGHeight  = 230.0f;                     // Gap left & right of box
    float Border    = 100.0f;

    // Position of the box
    FVector2D Pos(0+Border, Canvas->ClipY - BGHeight - 12.0f);

    // Draw the darkened box
    FCanvasTileItem BackgroundItem( Pos, FVector2D(BGWidth-Border*2,BGHeight) ,FLinearColor(0.0f, 0.0f, 0.0f, 0.8f));
    BackgroundItem.BlendMode = SE_BLEND_Translucent;
    Canvas->DrawItem( BackgroundItem );

    // Draw the text
    FCanvasTextItem TextItem( Pos + FVector2D(12.0f,12.0f), FText::FromString(HUDText), Font, FLinearColor(0.8f, 0.8f, 0.8f, 1.0f) );

    // Enable drop shadow
//    TextItem.FontRenderInfo.bEnableShadow = true;
//    TextItem.ShadowColor      = FLinearColor(0.0f,0.0f,0.0f,1.0f);
//    TextItem.ShadowOffset     = FVector2D( 1.0f, 1.0f );

    TextItem.BlendMode      = SE_BLEND_Translucent;
    TextItem.Scale          = FVector2D( 1.0f, 1.0f );

    Canvas->DrawItem( TextItem );
    return true;
}

// *****
// DrawProjectileText - Displays description text to show projectiles that are equipped
// *****
bool AFinalProjectRyanTHUD::DrawProjectileText()

```

```

{
    // Get a pointer to the player character
    AFinalProjectRyanTCharacter* Player = Cast<AFinalProjectRyanTCharacter>( GetWorld()->GetFirstPlayerController()->GetPawn()
);
    if ( !Player )
        return false;

    FString String = "Projectiles:";

    if ( Player->ProjectileClassIndex != eProjectileType::None )
    {
        String.Append( "\n" );
        String.Append( "    LButton: " );
        String.Append( Player->GetProjectileName(0) );
    }

    if ( Player->ProjectileClassIndexRight != eProjectileType::None )
    {
        String.Append( "\n" );
        String.Append( "    RButton: " );
        String.Append( Player->GetProjectileName(1) );
    }

    // Position of text
    FVector2D Pos(Canvas->ClipX - 240, 10);

    // Draw the text
    FCanvasTextItem TextItem( Pos, FText::FromString(String), Font, FLinearColor(0.8f, 0.8f, 0.8f, 1.0f) );

    // Enable drop shadow
    TextItem.FontRenderInfo.bEnableShadow = true;
    TextItem.ShadowColor      = FLinearColor(0.0f, 0.0f, 0.0f, 1.0f);
    TextItem.ShadowOffset     = FVector2D( 2.0f, 2.0f );

    TextItem.BlendMode       = SE_BLEND_Translucent;
    TextItem.Scale           = FVector2D( 1.0f, 1.0f );

    Canvas->DrawItem( TextItem );
    return true;
}

```

### *FinalProjectRyanTProjectile.h*

```
// Copyright 1998-2016 Epic Games, Inc. All Rights Reserved.
#pragma once
#include "GameFramework/Actor.h"
#include "FinalProjectRyanTProjectile.generated.h"

UENUM()
details panels
enum class eProjectileType : uint8
{
    Default,
    Sticky,
    Repel,
    Attract,
    None
};

UCLASS(config=Game)
class AFinalProjectRyanTProjectile : public AActor
{
    GENERATED_BODY()

    /** Sphere collision component */
    UPROPERTY(VisibleDefaultsOnly, Category=Projectile)
    class USphereComponent* CollisionComp;

    /** Projectile movement component */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
    class UProjectileMovementComponent* ProjectileMovement;

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    // Called when the game ends or when deleted
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

public:
    AFinalProjectRyanTProjectile();

    /** called when projectile hits something */
    UFUNCTION()
    virtual void OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector
NormalImpulse, const FHitResult& Hit);
```

```

/** Returns CollisionComp subobject */
FORCEINLINE class USphereComponent* GetCollisionComp() const { return CollisionComp; }
/** Returns ProjectileMovement subobject */
FORCEINLINE class UProjectileMovement* GetProjectileMovement() const { return ProjectileMovement; }

static TArray<class AFinalProjectRyanTProjectile*> ProjectileList;

eProjectileType      Type = eProjectileType::Default;
};

```

### *FinalProjectRyanTProjectile.cpp*

```

// Copyright 1998-2016 Epic Games, Inc. All Rights Reserved.

#include "FinalProjectRyanT.h"
#include "FinalProjectRyanTProjectile.h"
#include "GameFramework/ProjectileMovementComponent.h"
#include "DrawDebugHelpers.h"

TArray<class AFinalProjectRyanTProjectile*> AFinalProjectRyanTProjectile::ProjectileList;

AFinalProjectRyanTProjectile::AFinalProjectRyanTProjectile()
{
    // Use a sphere as a simple collision representation
    CollisionComp = CreateDefaultSubobject<USphereComponent>(TEXT("SphereComp"));
    CollisionComp->InitSphereRadius(5.0f);
    CollisionComp->BodyInstance.SetCollisionProfileName("Projectile");
    CollisionComp->OnComponentHit.AddDynamic(this, &AFinalProjectRyanTProjectile::OnHit);           // set up a notification for
when this component hits something blocking

    // Players can't walk on it
    CollisionComp->SetWalkableSlopeOverride(FWalkableSlopeOverride(WalkableSlope_Unwalkable, 0.f));
    CollisionComp->CanCharacterStepUpOn = ECB_No;

    // Set as root component
    RootComponent = CollisionComp;

    // Use a ProjectileMovementComponent to govern this projectile's movement
    ProjectileMovement = CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileComp"));
    ProjectileMovement->UpdatedComponent = CollisionComp;
    ProjectileMovement->InitialSpeed = 2500.f;
    ProjectileMovement->MaxSpeed = 2500.f;
    ProjectileMovement->bRotationFollowsVelocity = true;
}

```

```

ProjectileMovement->bShouldBounce = true;

// Die after 3 seconds by default
InitialLifeSpan = 3.0f;
}

void AFinalProjectRyanTProjectile::BeginPlay()
{
    Super::BeginPlay();
    ProjectileList.Add(this);           // Add this projectile to the projectile list
}

void AFinalProjectRyanTProjectile::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    Super::EndPlay(EndPlayReason);
    ProjectileList.Remove(this);        // Remove this projectile to the projectile list
}

void AFinalProjectRyanTProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveComponent* OtherComp,
FVector NormalImpulse, const FHitResult& Hit)
{
    // Only add impulse and destroy projectile if we hit a physics
    if ((OtherActor != NULL) && (OtherActor != this) && (OtherComp != NULL))
    {
        if (OtherComp->IsSimulatingPhysics())
            OtherComp->AddImpulseAtLocation(GetVelocity() * 100.0f, GetActorLocation());

#if 0
        DrawDebugLine(GetWorld(), Hit.Location, Hit.Location + Hit.ImpactNormal*100.f, FColor::Green, true, 5.0f, 0, 5.0f);

        FVector Direction = GetVelocity();
        Direction.Normalize();
        DrawDebugLine(GetWorld(), Hit.Location, Hit.Location - Direction*100.f, FColor::Red, true, 5.0f, 0, 5.0f);
#endif

        UE_LOG(LogTemp, Warning, TEXT("hit %s"), *OtherComp->GetName());
        Destroy();
    }
}

```