스폰서 기획: <저스트 코즈2>의 세계 – 대규모 오픈 랜드스케이프 구축을 위한 창의적인 테크놀로지

(Sponsored: The World of Just Cause2 – Using Creative Technology to Build Huge Open Landscapes)

작성자: 리누스 블롬버그 (Linus Blomberg)

작성일: 2013년 5월 16일

아발란쉬 스튜디오는 10년 전에 오픈 월드 게임에 대한 열정을 품고 시작된 회사이다. 그러나 나에게 있어서는 1984년 이안 벨(Ian Bell)과 데이빗 브라벤(David Braben)의 BBC 마이크로 모델B(BBC Micro Model B)용 고전 게임 '엘리트(Elite)'를 플레이 할 때 이미 시작되었다. 32KB의 메모리와 2MHz로 운영되는 CPU만으로, 자유롭게 탐험할 수 있는 8,000개의 각기 다른 행성이 있는 우주를 만들어냈다는 사실에 나는 경탄했었다. 당시의 플랫폼 게임이나 사이드크롤러(sidecroller)와는 완전히 다른 경험에 나는 넋을 잃고 말았다.

침실 스페이스의 우주조종사들

나는 두 동생들과 함께 아버지가 대학 교수로 재직 중 학교에서 가져다 주신 낡은 폐기계를 사용하여 집 안 옷장에다 우주선 조종실을 만들었다. 중앙에는 BBC 와 조이스틱이 있었다. 맏형인 내가 당연히 가장 높은 계급으로 조이스틱을 맡았고, 동생들은 유도방향전환장치에 손가락을 두고 적의 미사일이 공격해 오면 즉시 긴급이동하도록 명령했다. 최악의 경우에는 탈출 작전을 펼쳐야 했다. 너무 늦게 누르면 진공의 우주공간-옷장 밖 침실-으로 빨려 들어가는 것이었다. 이렇게 심한 벌을 주었는데도 내가 운이 좋았는지 우리 형제들은 오픈 월드 게임에 대한 열정을 잃지 않았고, 결국 직접 게임을 만들어 보기에 이르렀다.

우리는 곧 프랙탈(fractal)을 발견했고 절차적(procedural) 테크닉으로서 하드웨어상의 전형적인 한계를 극복할 수 있다는 것을 깨달았다. – 몇 년 전 브라벤(Braben)과 벨(Bell)이 발견한 사실을 우리도 깨달은 것이다. 스웨덴 북부의 시골에 살고 있던 우리는 이 기술을 활용하여 창문 밖의 랜드스케이프를 묘사하고 싶어졌다. 수년에 걸친 수많은 코딩 작업 끝에 절차적인 랜드스케이프의 데모가 완성되었고, 에이도스(Eidos)의 한 프로듀서가 관심을 보여 <저스트 코즈(Just Cause)> 시리즈가 탄생하였다.

우주공간 저 너머로

<저스트 코즈> 시리즈 게임플레이의 근본은 거대한 세계에서 자유롭게 움직이는 능력이다. 또한 <저스트 코즈>의 경험에서는 원거리 표현(long draw distance)이 매우 중요하다. 플레이어의 프로그레션과 모티베이션 대부분을 비주얼 인풋으로 안내하기 때문이다. 우리는 플레이어가 수평선에서 무엇인가를 보고 저 먼 산 뒤에나 머나먼 섬에 무엇이 숨어있는지 알고 싶어하기를 바랐다. 아발란쉬 엔진은 처음부터 원거리 표현이 필요한 거대한 오픈 월드의 샌드박스 게임플레이를 위해 개발된 엔진이다.

이를 달성하기 위해서는 많은 기술이 필요하기 때문에, 기존의 게임 엔진에 오픈월드 기능을 추가하는 것은 매우 어려운 일이다. 이 글에서는 '지형 렌더링(terrain rendering)'이라고 하는 기술을 소개하고자 한다. 특히 <저스트 코즈>게임의 경우 어떻게 '지형 메쉬(terrain mesh)'가 생성되는지, 32*32 킬로미터의 세계를 모두 가시거리로 그려냈는지 설명할 것이다. 우리의 지오-모핑(geo-morphing) 테크닉으로 가시거리를 길게 하면서 동시에 'LOD 팝핑(level-of-detail popping)'을 제거한 과정, 리소스 관리 테크닉과 지형 메쉬의 생성에 필요한 방대한 데이터를 효과적으로 스트리밍하고 재창조한 데이터 압축에 대해서도 소개하려고 한다.

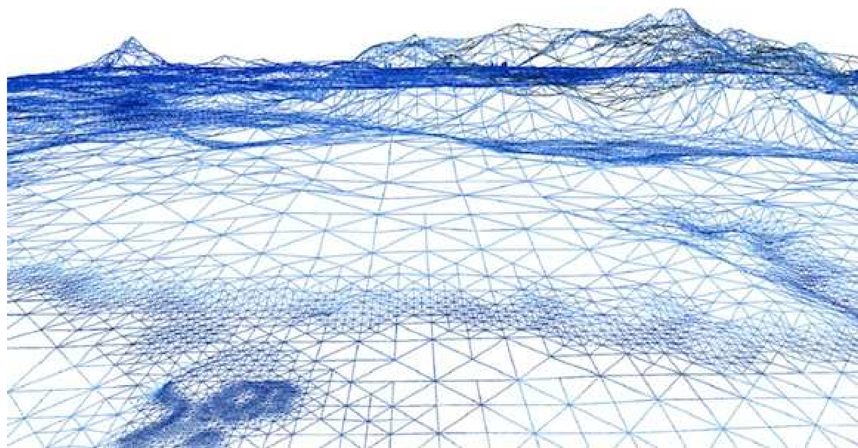
아발란쉬 엔진에서 지형 시스템을 디자인하는 목표는 다음과 같다.

메모리 효율성. 오리지날 <저스트 코즈>는 플레이스테이션 2 에서 출시되었기 때문에 우리는 이렇게 큰 월드를 구현하기에는 할당 메모리가 부족하다는 것을 예상했다. 데이터 압축과 절차적 테크닉이 필요했다.

고성능. 오픈월드 게임에서는 지형 시스템이 프로세서를 지나치게 소비하지 않도록 하기 위해 렌더링이 많이 필요하다. 이를 위해서 스태틱 버텍스 버퍼(static vertex buffer)를 사용해야 한다는 것은 알고 있었는데, 이 경우 일부 지형 렌더링 테크닉은 배제될 것이었다.

하이파이 비주얼. 이는 고해상도, LOD 팝핑의 제거, 긴 거리 묘사 등을 말한다. 다시 한 번 강조하면, 다른 두 목표를 달성하기 위해서도 절차적 테크닉은 필수적인 기술이다.

<저스트 코즈2>의 전형적인 풍경



같은 풍경을 와이어프레임으로 본 장면.
해안가나 복잡한 지역에서는 해상도가 높아지는 것을 볼 수 있다.

패치의 개념

아발란쉬 엔진에서 랜드스케이프 시스템의 핵심은 "패치"의 개념이다. 여기서 '패치'는 2 차원의 데이터 컨테이너 공간을 말한다.

패치의 크기는 언제나 2 차원 두 선의 제곱으로 나타나며, 포지션은 항상 자신의 사이즈의 몇 배가 되어 월드 액스(world axes)로 정렬된다. 패치에 들어 있는 실제 컨텐츠가 그 타입을 결정한다. 우리는 엔진에서 스트림(stream) 패치, 지형(terrain) 패치, 베지테이션(vegetation) 패치 등 여러가지 패치 타입을 사용한다. 이 각각의 패치 타입에 고유의 사용 목표가 있는 것이다.

같은 타입의 N*N 패치들의 모음을 '패치 맵(patch map)'이라고 한다. 보통 패치 맵은 카메라의 중심에 위치한다. 즉 카메라가 움직이면 패치 맵의 일부 패치는 사라지고 다른 패치가 창조되는 반면, 일부 패치는 계속 살아 있게 된다는 뜻이다.

패치 맵에는 또한 '패치 시스템'이라고 하는 위계가 존재한다. 패치 시스템이란 같은 타입의 패치 맵의 모음을 말한다. 각각의 패치 맵에는 같은 수의 패치가 포함되어 있지만 패치들간의 사이즈는 제곱으로 차이가 난다. 이 구조를 이용하여 같은 데이터에서 서로 다른 LOD 를 나타낼 수 있다. 우리는 이러한 위계로 되어 있는 패치들을 부모-자식 관계라고 부른다. 같은 공간에 펼쳐져 있다는 것을 제외하면 실제로 관련성은 거의 없지만 말이다.

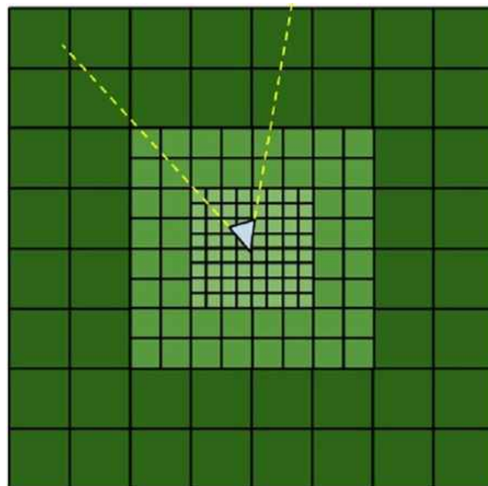


그림 1: 카메라 중심부의 3단계 패치맵의 패치 시스템

중앙 패치 관리자가 이 패치들의 투입과 조직을 관리한다. 패치 관리자는 패치의 타입은 알지 못한다. 패치 헤더(header)와 알려지지 않은 데이터의 작은 블롭(blob)만 볼 수 있을 뿐이다. 이후 등록된 콜백(callback)에서 패치의 창조나 삭제를 비롯하여 각 타입에 맞는 액션을 취한다. 이 추상적 개념으로서 코어 패치를 더 쉽게 최적화할 수 있으며, 동시에 수천가지 패치를 운영할 수 있는 매우 유용한 방법이기도 하다. 패치 관리자는 또한 뷰 절두체(view frustum)와 대비하여 패치를 테스트하거나 메모리 관리를 하는 등 일반적인 작업들도 수행한다.

데이타 파이프라인

아발란쉬 엔진에서는 JustEdit(이름 참 좋다!)이라는 특허 편집기를 사용하여 지형을 제작한다. 이 단계에서는 데이터의 크기를 신경 쓸 필요가 없다. 개발자의 워크스테이션은 RAM 이 충분하기 때문이다. 따라서 JustEdit 에서 지형을 제작할 땐 개략적인 하이트 맵(height map) 포맷과 고정 해상도 프록시 메쉬(fixed resolution proxy mesh)를 사용한다. 그러나 성능을 높이기 위해 거리에 따라 패치별로 해상도를 다르게 하기도 한다. JustEdit 에서 지형 데이터는 디스크에서 에디터 패치(editor patches)로 쪼개져서 각각이 퍼포스(Perforce)에 체크인 및 체크아웃을 할 수 있고, 따라서 다수의 아티스트들이 동시에 월드 작업을 할 수 있게 된다.

이렇게 생성된 데이터는 데이터 포맷을 특정 플랫폼에 맞게 최적화하고 *스트림 패치*로서 출력하는 지형 컴파일러로 전송된다. 전체 월드를 다 이렇게 작업하면 매우 시간이 많이 걸리므로, 주로 끊임없이 개선이 필요한 구역 위주로 사용한다. 우리는 로컬 지역의 라이브 편집을 위해 최신 툴인 GPGPU 프로세스를 사용하고 있다.

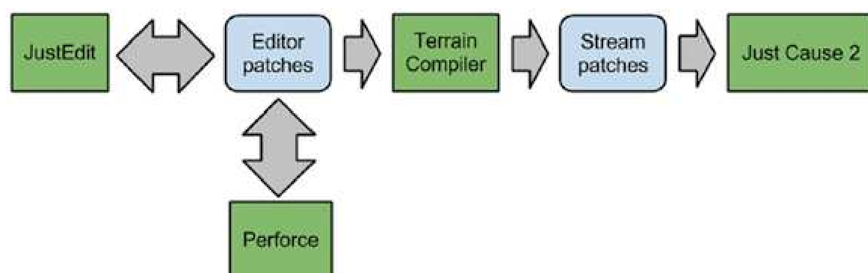


그림 2: 데이터 파이프라인

스트리밍(Streaming)

<저스트 코즈>시리즈에서는 플레이어의 움직임에 제한이 없다(스피드는 빼고). 따라서 우리는 플레이어가 움직이는 방향과 상관 없이 효과적으로 스트림-인(streamed-in)될 수 있는 방식으로 디스크에 런타임 데이터를 구성해야 했다. 이를 위해서 전체 월드를 64*64 셀들로 구성된 규칙적인 격자무늬로 나누었다. 월드 사이즈가 32*32 킬로미터이므로, 각각의 셀에는 512*512 미터가 할당된다. 우리는 각각의 셀을 스트림 패치(stream patch)라고 부르는데, 이 셀들이 해당 구역을 렌더링하고 물리적으로 나타내기 위해 필요한 모든 지형 데이터를 담고 있다. 우리는 메모리 내에 카메라를 중심으로 8*8 스트림 패치의 패치 맵을 유지하고 있다.

우리는 데이터를 옵티컬 디스크에서 바로 스트리밍할 수 있기를 원했기 때문에 데이터를 읽는 데 필요한 검색(seek) 횟수를 최소한으로 유지하는 것이 매우 중요했다. 하지만 디스크 사이즈는 쉽게 확보할 수 있었고, 이를 활용하여 실제로 스트림 패치를 두 번(x/z 오더와 z/x 오더로 각각) 저장하였다. 이렇게 해서 8 개의 스트림 패치의 새로운 열이나 행을 불러오는 데 한 번의 검색이면 충분했다. 그래서 최대 트래블 속도(travel speed)는 옵티컬 드라이브(optical drive)의 데이터 전송률로 제한되었다.

결국 우리는 게임에서 움직임의 최대 가능 속도를 수평 방향으로 초당 512 미터로 제한하기로 했고, 각각의 스트림 패치의 평균 데이터 크기는 약 128kb 가 되었다. "평균"이라고 하는 이유는 각 패치의 데이터 크기가 고정된 것이 아니었기 때문이다. 더 중요한 것은 임의로 위치에서 8*8 스트림 패치를 조합했을 때 데이터가 얼마나 들어 있느냐였다. 이렇게 해서 일부 패치에서 평균 이상의 데이터를 포함하는 것도 가능하게 되었다. 이를 둘러싼 패치들의 데이터 사이즈가 평균 이하라면 가능했다. 메모리가 가능한 모든 위치를 수용할 수 있게 하기 위해서 지형 컴파일러의 패스(pass)가 세계 전체에 걸쳐 이터레이션(iterate)을 수행한다.

각 스트림 패치에 저장된 주요 지형 데이터 분류는 다음과 같다.

- 지형 텍스처 (머테리얼 맵(material map)과 노멀 맵(normal map))
- 하이트 맵(Height map)과 머테리얼 맵 (16 비트/샘플로 압축되어 있음)
- 지형 메쉬 데이터 (Terrain mesh data)

터레인 텍스처(The Terrain Textures)

우리는 <저스트 코즈 2>의 지형을 렌더링하는데에 하이트 맵을 외에도 다음 3 가지 타입의 텍스처를 사용한다.

- 노멀 맵 (A8L8 포맷) – 지형의 노멀
- 머테리얼 지표 (ARGB4444 포맷) – 머테리얼 타입 인덱스
- 머테리얼 웨이트(ARGB4444 포맷) – 각 머테리얼 타입의 웨이트(weight)

노멀 맵 텍스처는 당연히 셰이딩(shading)에 사용된다. – 여기서는 특별할 것이 없다. 버텍스 대신 텍스처에 노멀 맵 정보를 저장하면 라이트닝(lightning)이 메쉬 해상도에 따라 변하지 않는다는 장점이 있다. 이는 특히 LOD 작업을 숨길 때 크게 도움이 된다. 머테리얼 텍스처는 픽셀 셰이더에서 다양한 고해상도 재료의 텍스처 맵들 간에 선택하거나 혼합하는 데 사용된다. 각각의 텍스처의 해상도는 텍셀(texel)당 4 미터이다. 이는 지형 텍스처로서는 매우 낮은 해상도이지만, 우리는 최종적으로 고해상도를 달성하기 위해서 모듈레이션(modulate)이 가능한 타일(tiled) 형태의 디테일 텍스처(detail texture)를 사용한다. 이것이 우리가 데이터의 충실도를 높이기 위해 절차적 테크닉을 사용하는 한 사례이다.

이 글에서는 픽셀 셰이더에 대한 세부 설명은 생략하겠다. 분명한 것은 픽셀 셰이더가 비용이 많이 들고 최적화를 위해 많은 시간이 걸렸다는 사실이다. 사실은 때때로 화면에 나무가 많을 때 프레임률(frame rate)이 높아지기도 하는데, 이는 나무들이 지형을 가리기 때문이다. 그래서 우리는 비싼 지형 픽셀은 렌더링을 자제하는 편이다. 드로잉(drawing)을 더 많이 할수록 성능도 개선된다는 것은 매우 흥미로운 사실이다.

하이트 맵(Height Map)과 머테리얼 맵(Material Map)

JustEdit 에서 하이트 맵(Height Map)과 머테리얼 맵(Material Map)의 소스 데이터의 해상도는 샘플당 4 미터이다. 하이트는 16 비트 값으로 저장되고, 머테리얼 인덱스는 8 비트로 저장된다. PC 버전의 <저스트 코즈 2>에서는 이 포맷을 유지하고 있지만, Xbox 360 과 플레이스테이션 3 버전에서는 맵을 전체 샘플당 16 비트로 압축하고 있다. 이 압축은 하이트와 머테리얼 샘플의 4*4 블록들(blocks)이 필요하고 데이터에 수많은 단순화를 해야 하는, 매우 “로시한(lossy, 손실이 많은)” 방식이다. 머테리얼은 4 엔트리(entry)의 테이블을

사용하고 테이블의 룩업(lookup) 지표로 사용되는 각 샘플에 2 비트 값을 사용한다는 점에서 DXT 텍스처 포맷과 비슷한 방식으로 단순화된다.

하이트값은 3:6 플로팅 포인트(floating-point) 포맷으로 전환되며, 여기서 3 비트 지수는 2*2 샘플과 공유되고 각 샘플은 6 비트 맨티서(mantissa)를 갖는다. 플로팅 포인트 형태를 사용하면 필요한 경우 주파수가 낮은 지역에서 정확도를 높일 수 있다. 블록에는 또한 레이-캐스트(ray-cast)를 최적화하기 위한 바운딩 데이터(bounding-data)가 팩(pack)되어 있다. 런타임으로 데이터를 샘플링할 때는 압축해제된 샘플들에 카트뮬-롬 스플라인(Catmull-Rom spline) 보간법(interpolation)이 사용되며, 머테리얼 맵으로 콘트롤 되는 고해상도 디스플레이스먼트 맵들(displacement maps)이 결과에 추가된다. 이렇게 해서 상대적으로 적은 양의 데이터로 충실도 높은 하이트 값을 산출할 수 있는 것이다.

하이트 맵은 피직스 시스템(physics system)으로 프레임당 수천번도 샘플링될 수 있기 때문에, 샘플링 기능에서 고성능을 내는 것이 아주 중요하다. 그래서 우리는 데이터를 언팩(unpack)하고 샘플링하는 데 수동으로 튜닝한(hand-tuned) SIMD 기능을 사용한다. Xbox 360 과 플레이스테이션 3 의 VMX 인스트럭션은 매우 강력했고, 대부분 다용도의 퍼뮤트(permute) 인스트럭션 덕분에 목표를 이룰 수 있었다. 불행히도 이 인스트럭션은 x86 아키텍처(architectures)에는 존재하지 않기 때문에 우리는 PC 에서는 다른 종류의 데이터 표현 방식을 써야 했다.

터레인 메쉬(Terrain Mesh)

지금까지 스트림 패치들이 어떻게 512*512 미터의 고정된 사이즈를 갖게 되는지 알아보았다. 스트림 패치가 단순히 다른 시스템에서 요청이 있을 때 데이터를 제공하기 위해 스트리밍 되는 데이터 컨테이너에 불과하다고 생각하는 사람도 있을 것이다. 이제 역시 패치 종류로 표현되는 지형 메쉬에 대해 살펴볼 것이다. 우리는 이들을 '지형 패치(terrain patches)'라고 부르는데, 기본적으로 런타임으로 생성되며 해당 지역의 지형을 나타내는 버텍스(vertex)와 인덱스 버퍼(index buffer)를 담고 있다.

지형 패치는 패치 시스템(예를 들어 위에 설명한 일련의 패치 맵들)으로 조직된다. 이는 지형 메쉬에 몇 가지 LOD 표현 방식이 있기 때문이다. <저스트 코즈 2>에는 12 단계의 지형 메쉬 디테일이 있다. 각 단계는 카메라를 중심으로 8*8 지형 패치의 패치 맵으로 되어 있다. 가장 작은 단계의 지형 패치 맵이 전체

64*64 미터의 구역을 커버하며, 가장 큰 단계의 경우 256*256 킬로미터까지 커버한다. 이 사이즈가 게임 월드보다 훨씬 큰 규모라는 사실에 주목하자. 우리는 플레이어가 맵에서 어디에 있든지 관계 없이 한 방향에서 보이는 거리가 일정하도록 만들기 위해 게임 월드 밖에서 데이터를 절차적으로 생성하였다.

카메라를 중심으로 한 패치의 LOD 피라미드 개념은 휴 호프와 프랑크 로사소(Hugues Hoppe and Frank Losasso)가 묘사했던[1] 클립맵(Clipmaps)과도 유사하지만, 우리는 사실 2004 년 이 보고서가 나오기 전부터 우리의 시스템을 개발했다.

카메라가 움직임에 따라 새로운 지형 패치의 열과 행이 조합되고 기존의 열과 행은 삭제된다. 이 데이터들은 스트림 패치에서 수집했거나 지형 패치의 LOD 수준에 따라 글로벌 로우-피델리티 데이터 표현(global low-fidelity data representation)으로부터 수집한 것이다.

컴파일-타임 메쉬의 조직(Compile-Time Mesh Construction)

각각의 지형 패치에는 그 지역의 지형을 나타내는 메쉬(mesh)가 포함되어 있다. 이 메쉬의 해상도는 균일하지 않다. <저스트 코즈>시리즈의 랜드스케이프는 로케이션별로 다양하고 복잡하기 때문에, 그 지역을 정확하게 나타내기 위해 필요한 트라이앵글(triangles) 수도 그만큼 제각각이다. – 따라서 우리는 균일한 지형 메쉬 해상도를 포기하기로 했다. 불필요한 버텍스 때문에 버텍스 성능과 메모리를 낭비하고 싶지 않았기 때문이다. 또한 작은 트라이앵글은 큰 트라이앵글에 비해 렌더링이 픽셀당 더 비싸기 때문에, 작은 트라이앵글은 시각적으로 큰 효과가 있는 경우에 한해서만 사용하기로 했다. 그러나 이는 그래픽 표현에만 해당하는 이야기다. 저변의 물리적인 표현을 위해서는 위에서 언급한 고정 사이즈의 하이트 맵을 사용했다.

뷰포인트(viewpoint)를 고려하여 해상도를 실시간으로 조정할 수 있는, 해상도 조절가능 메쉬 스킴 (adaptive-resolution mesh scheme)도 많이 나와 있다. "리얼타임 최적 조정 메쉬(real-time optimally adapting mesh)" (ROAM)이 한 예이다. 이 때 한 가지 문제점은 이들이 뷰포인트를 바꿀 때 "팝핑" 부산물을 만드는 경우가 많다는 것이다. 더 중요한 문제는 버텍스 카운트(vertex count)가

[1] 참조 링크: http://research.microsoft.com/en-us/um/people/hoppe/geomclipmap.pdf

프레임마다 달라질 수 있다는 점이다. 특히 구형 하드웨어에서는 각 프레임마다 새로운 버텍스를 업로드해야 하기 때문에 매우 비효율적이 된다.

이러한 문제 때문에 우리는 지형 패치의 수명이 다할 때까지 고정된 스태틱 버텍스 버퍼(fixed static vertex buffer)를 사용해야 했다. 이는 우리가 기본적으로 뷰포인트를 무시하고 지형의 복잡성과 카메라로부터의 거리에만 중점을 두고 해상도를 결정했다는 것을 의미한다. 지형 복잡성 부분에서는 우리는 오프라인 메쉬 조합 솔루션(offline mesh construction solution)을 택했고, 거리 부분에서는 같은 데이터에 대해 각기 다른 LOD 표현의 런타임 셀렉션(run-time selection)을 사용하였다.

메쉬 데이터 표현을 위해서 우리는 바이너리 트라이앵글 트리(binary triangle tree)에 기반한 적응기법(Adaptive Scheme)을 생각해 냈다. 우리는 독자적으로 고안한 것이지만, 현재 인터넷에 보면 바이너리 트라이앵글 트리에 대한 몇 가지 참고 자료들이 나와 있다. 우리의 접근 방식과 비슷한 사례가 2006 년 크리스 달레어(Chris Dallaire)가 썼던 가마수트라 아티클[2]에도 소개되어 있다. 바이너리 트라이앵글 트리는 간단히 설명하면, 한 노드가 한 트라이앵글을 정의하고 두 개의 하위 노드가 상위 트라이앵글의 일관된 하위구분을 정의하는 방식의 트리를 생각하면 된다(그림 3 참조). 사각형의 지형 패치를 표현하기 위해서는 두개의 트라이앵글 트리가 필요하다. 사각형의 패치는 대각선을 따라 쪼개지고, 결과 두 개의 트라이앵글은 두 트라이앵글 트리의 루트 노드(root nodes)가 된다.

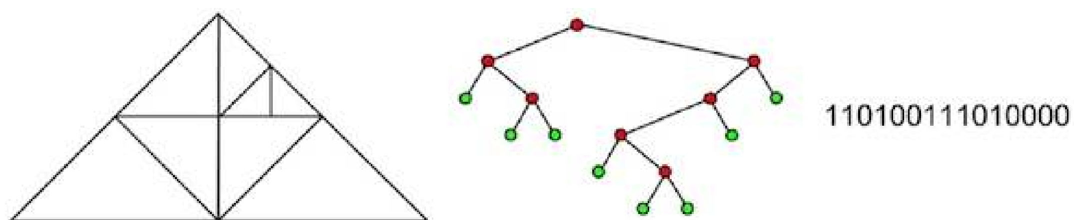


그림 3: 같은 트라이앵글 트리의 세 가지 다른 표현: 메쉬, 트리, 비트 스트림(bit stream)

지형 컴파일러에서 오프라인 프로세스는 지형 전체에 걸쳐 이터레이션을 하고 지형학적 복잡성을 측정함으로써 이 트라이앵글 트리들을 생성한다. 지형이 복잡할수록 결과 생성된 트라이앵글 트리는 더 깊어진다.

[2] 참조 링크: http://www.gamasutra.com/view/feature/1754/binary_triangle_trees_for_terrain_.php

우리는 또한 트리의 깊이에 영향을 주는 복잡성 외에 휴리스틱스(heuristics, 편리한 기준에 따라 일부만을 고려하여 문제를 해결하는 방법)도 사용했다. 예를 들어 섬의 가장자리를 부드럽게 만들기 위해서 해안가에는 언제나 최상의 해상도를 사용했다. 이는 루트 노드로 시작해서 그 트라이앵글 내에서 하이트 맵으로 트래버스(traverse)하여 트라이앵글 면(plane)과 하이트 맵 샘플간의 차이를 합산하는 분석 작업이다. 이 합계치가 특정 기준을 넘으면 트리는 다시 하위에 분류되어 같은 과정이 하위 노드에서 반복된다. 결과 생성된 트리는 "1"이 한 노드를 나타내고 "0"이 잎(leaf)을 나타내는 하나의 비트-스트림(bit-stream)으로 저장된다. 이는 지형 메쉬를 매우 압축한 표현이다. X 와 Z 포지션들은 트리 구조에 내포되어 있고, 각 노드의 하이트값은 버텍스 버퍼가 생성될 때 하이트 맵으로부터 런타임으로 샘플링 된다.

오리지널 <저스트 코즈>게임에서는 사실 이 구조에서도 각 노드에 하이트들을 저장했었고, 트라이앵글 내에서 하이트 맵 값을 생성하기 위해 평면 보간법(planar interpolation)을 사용하였다. 물론 이는 하이트 맵을 저장하는 압축적인 방법이지만, <저스트 코즈 2>에서는 단순성을 더 중시하여 이 방법을 쓰지 않기로 했다. 더 이상 플레이스테이션 2 와 오리지널 Xbox 를 지원하지 않기 때문에 메모리의 여유는 충분했다.

결국 핵심은 지형 메쉬와 관련하여 각 스트림 패치에 저장된 정보는 단순히 각 트라이앵글 트리를 위한 비트-스트림이라는 것이다. 그러나 문제가 조금 복잡해지는 이유는, 우리가 사실상 지형 패치 맵에서 이 과정을 각 패치 사이즈마다 한번씩 수행하기 때문이다. 우리는 기본적으로 지형 메쉬를 각각 다른 해상도로 12 번 넘게 저장한다. 지형 패치 시스템에 12 개의 패치 맵이 있기 때문이다. 스트림 패치보다 작거나 비슷한 패치 사이즈에 맞는 메쉬들은 스트림 패치에 저장되고 스트림 패치 맵의 일부로서 스트림 되지만, 스트림 패치보다 큰 패치 사이즈에 맞는 메쉬들은 "항상 로드되는(always loaded)" 글로벌 리포지터리(global repository)에 저장된다.

지오모핑(Geomorphing)

이제 우리는 지형 패치 시스템에 조직된 지형 메쉬의 12 가지 디테일 표현을 갖게 되었다. 이들 모두를 한꺼번에 다 사용할 필요는 없다. 그렇게 하는 건 바보같은

방법이고, 보기에도 좋지 않다. 카메라로부터의 거리에 기반하여 런타임으로 골라야 한다.

또한 우리는 기존 셀렉션 부분과 새로운 셀렉션 사이를 이동할 때 "팝핑"이 일어나지 않도록 블렌딩 방법을 찾아야 했다. 특정 블렌드 지역에서 사이 구역을 픽셀셰이더로 알파 블렌드(alpha blend)하는 것도 한 가지 방법이다. 그러나 이렇게 하면 두 메쉬간의 차이가 클 경우 깨짐 현상이 나타날 수 있다.

결국 우리는 지오모핑(geomorphing)을 이용하기로 했다. 지오모핑이란 점차적으로 "모핑(morphing)," 즉 조정하는 것을 말하는데, 특정 트랜지션 범위를 따라서 거리에 기반하여 더 높은 해상도쪽 메쉬의 버텍스 포지션을 낮은 해상도 메쉬에 맞춰 조정하는 것이다. 이 작업이 깨짐 현상 없이 성공하기 위해서는 모핑 타겟이 낮은 해상도쪽 메쉬의 가장자리와 정확하게 맞아야 한다.

이 기술은 사실 서로 다른 LOD 메쉬의 가장자리를 따라 T-연결지점(junction)을 만들게 된다는 것을 알아야 한다. 그러나 실제로 해 보면 이 문제는 크게 눈에 띄지 않았다.

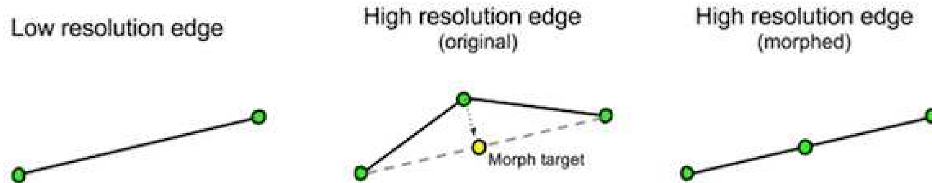


그림 4: 고해상도 메쉬의 가장자리를 저해상도 메쉬와 맞추어 모핑하는 과정

트라이앵글 트리를 쓰면 사실상 추가 정보를 저장할 필요 없이 이 모핑을 자연스럽게 할 수 있다. 더 높은 해상도의 트라이앵글 트리를 포함하는 하위 지형 패치가 더 저해상도의 트라이앵글 트리를 포함하는 상위 지형 패치를 부분적으로 커버하고 있다고 해보자. 저해상도 트리와 고해상도 트리를 동시에 순회(traverse)한다면, 우리는 쉽게 저해상도 트리에 기반하여 고해상도 트리를 위한 모핑 타겟 정보를 생성할 수 있다. 이 방법이 효과적인 이유는 고해상도 트리는 항상 같은 구역의 저해상도 트리의 확대집합(superset)이기 때문이다.

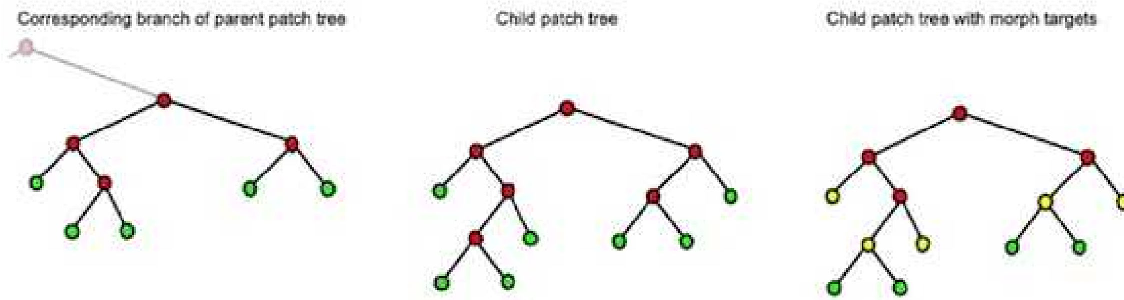


그림 5: 고해상도 트리와 병행하여 저해상도 트리를 순회하면 모핑 타겟을 정할 수 있다.

버텍스(Vertex)와 인덱스(Index) 버퍼의 생성

버텍스와 인덱스 버퍼의 생성 방법은 간단하다. 지형 시스템이 패치 매니저로부터 지형 패치에 대한 컨스트럭션 콜백을 받으면 해당 지형 패치의 트라이앵글 트리 데이터를 위한 해당 스트림 패치에 쿼리(queries)를 수행한다. 그리고 나서 스트림 패치에 상위 지형 패치의 트라이앵글 트리 데이터도 요청한다. 이제 상위 지형 패치가 하위 패치의 4 배나 되는 구역을 커버하므로 같은 사분면(quadrant)를 커버하는 해당 트리의 가지(branch)를 하위 패치로 위치시켜야 한다. 그러면 두 트라이앵글 트리가 동시에 순회하며 버텍스와 인덱스 버퍼를 만들고, 상위 트리는 모핑 타겟을 지정하는 데 사용된다. 다음 가상 코드는 이 방법을 나타낸 것이다

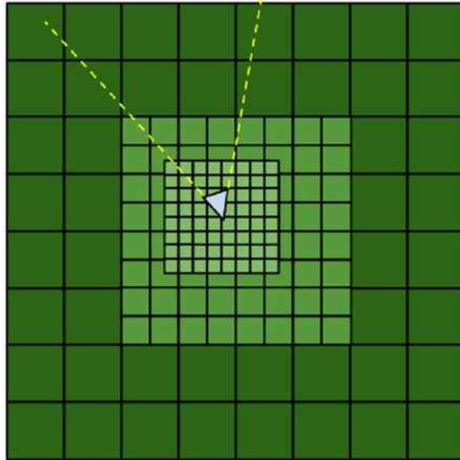
```
TraverseTree(vector2 p0, vector2 p1, vector2 p2)
{
    if child_bitstream.pop() == 1             // Node
    {                                         // Create new vertex
        new_vertex = (p0+p2)/2                // New vertex is the midpoint of the hypotenuse

        if parent_bitstream.pop() == 1        // Update morph target if node present in parent
            morph_target = new_vertex

        if !VertexBuf.exist(new_vertex)       // Only add if not already added by neighbor
            VertexBuf.add(new_vertex, morph_target)

        TraverseTree(p1, new_vertex, p0)      // Recurse to left child triangle
        TraverseTree(p2, new_vertex, p1)      // Recurse to right child triangle
    }
    else                                      // Leaf
        IndexBuf.add(p0.idx, p1.idx, p2.idx)  // Add triangle to index buffer
}
```

그림 6: 지형 메쉬의 버텍스와 인덱스 버퍼를 구축하는 순회 트리 기능을 나타낸 가상 코드

패치를 부분적으로 오버랩(Overlapping)하기

당연한 결과지만, 패치 시스템에는 하위 패치로 완전히 커버되는 패치도 있고, 부분적으로만 하위 패치로 커버되거나 전혀 커버되지 않는 패치도 있다. 이는

각각의 패치 맵이 카메라를 중심으로 위치하여 카메라가 이동할 때마다 다른 해상도로 '움직이기' 때문이다. 패치들간의 크기가 각 패치 맵 사이에서 제곱으로 차이가 나기 때문에 나타나는 현상이다.



결국 우리는 마지막 장애물이라고 할 수 있는 부분적으로 커버된 지형 패치 문제를 해결해야 한다. 패치의 각 사분면을 분리하여 그릴 수 있도록 하기 위해서이다. 이는 각 패치의 인덱스 버퍼를 사분면 안으로 정리함으로서 어떤 사분면이라도 독립적으로 그릴 수 있도록 하면 해결된다. 각 패치를 위한 하위 커버리지 마스크(coverage mask)이 패치 시스템에서 제공되어 어떤 사분면들이 하위 패치로 폐쇄(occluded)되어 있는지 알려준다.

결론

아발란쉬 엔진의 지형 시스템은 다양한 하드웨어에서 효과적이라는 것이 입증되었다. 이는 수년간 개발 과정을 반복한 결과이며, 아직 기밀이라서 자세한 내용을 밝힐 수는 없지만 <저스트 코즈 2>의 출시 이래로 더욱 눈부신 발전을 했다. 대부분의 디자이너들이 꿈도 꾸지 못했던 게임 디자인을 가능하게 하는 테크놀로지를 창조하는 이 일이야말로 벽장에서 우주선 놀이를 했던 어린 시절부터 끊임없이 나에게 영감을 주고 있는 원동력이다.

1984 년에는 대부분의 게임들이 플랫폼에서 괴물들을 피하면서 점프하는 것이 고작이었다. 그 시절 처음으로 8,000 개의 독특한 행성에서 트레이더, 해적, 현상금 사냥꾼 등으로 자유롭게 변신할 수 있는 게임 아이디어를 생각해 낸 사람들이 있었다. 브라벤(Braben)과 벨(Bell)이 바로 그들이다. 이들은 창의적인 테크놀로지의 마력을 알고 있었던 것이다. 새로운 시장이 떠오르고 신세대

하드웨어가 눈부시게 발전하고 있는 오늘날, 당신이 30 년 후에 사람들이 기사에서 참고로 읽게 될 테크놀로지를 개발 중이라면 하루하루가 정말로 즐거운 시간들일 것이다.

©2013, UBM LLC. Initially published in *Gamasutra.com*, a UBM LLC publication. Translated with permission.