

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM**

❖ NGÔ THÁI AN 0112085

❖ NGUYỄN ĐÌNH TOÀN 0112287

NGHIÊN CỨU VÀ XÂY DỰNG THỬ NGHIỆM 3D ENGINE

KHÓA LUẬN CỬ NHÂN TIN HỌC

GIÁO VIÊN HƯỚNG DẪN

T.S DƯƠNG ANH ĐỨC

Th.S TRẦN MINH TRIẾT

NIÊN KHÓA 2001-2005

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn Khoa Công nghệ Thông tin, Trường Đại học Khoa học Tự nhiên Thành phố Hồ Chí Minh đã tạo điều kiện cho chúng em thực hiện đề tài tốt nghiệp này.

Chúng em xin chân thành cảm ơn thầy Dương Anh Đức và thầy Trần Minh Triết đã tận tình hướng dẫn, chỉ bảo chúng em trong suốt thời gian làm đề tài.

Chúng em cũng xin cảm ơn quý Thầy Cô trong Khoa đã tận tình giảng dạy, trang bị cho chúng em những kiến thức cần thiết trong suốt quá trình học tập tại trường. Chúng em xin gửi lòng biết ơn sâu sắc đến ba, mẹ, các bạn bè đã ủng hộ, giúp đỡ, động viên em trong suốt quá trình học cũng như thời gian làm luận văn đầy khó khăn, thử thách.

Mặc dù đã rất cố gắng hoàn thành luận văn với tất cả nỗ lực của bản thân, nhưng chắc chắn luận văn không tránh khỏi những sai sót và hạn chế, kính mong sự thông cảm, chỉ bảo của quý Thầy Cô và các bạn.

Nhóm thực hiện

Ngô Thái An và Nguyễn Đình Toàn

Tháng 7 năm 2005

LỜI MỞ ĐẦU

Ngày nay, các sản phẩm giải trí mà đặc biệt là Game đã mang lại một nguồn lợi nhuận to lớn cho ngành công nghiệp máy tính. Do nhu cầu của thị trường tăng nhanh, các Game ngày càng có chất lượng càng cao và thời gian xây dựng ngày càng được rút ngắn. Các Game 3D trên thị trường hiện nay rất đa dạng về nội dung và chủng loại nhưng cùng có điểm chung là xây dựng trên các Game Engine. Chất lượng của Game sẽ phụ thuộc vào chất lượng của Game Engine mà nó sử dụng.

Game Engine chính là phần cốt lõi để xây dựng Game hiện nay và 3D Engine chính là phần quan trọng nhất của Game Engine. Việc sử dụng Game Engine để xây dựng Game là một xu thế tất yếu để có những Game chất lượng cao trong thời gian ngắn. Tiếc thay, hiện nay ở nước ta việc xây dựng Game 3D cũng như Game Engine vẫn còn là điều mới mẻ. Chính vì vậy, chúng em thực hiện đề tài “**Nghiên cứu và xây dựng thử nghiệm 3D Engine**” với mong muốn góp vào những viên gạch đầu tiên để xây dựng nên ngành công nghiệp Game của nước nhà trong tương lai.

Là những người đi sau và được thừa hưởng những thành tựu từ quá trình phát triển công nghệ thế giới, chúng em đã áp dụng những kỹ thuật mới nhất hiện nay vào trong 3D Engine của mình. Chúng em đã xây dựng nên **Nwfc Engine** là một 3D Engine với chức năng chính là dựng hình và quản lý tập tin. Sau đó chúng em tìm hiểu và xây dựng thêm một số hệ thống khác như hệ thống quản lý diễn hoạt, hệ thống vật lý, hệ thống hiệu ứng (particle và âm thanh) để kết hợp với 3D Engine Nwfc tạo ra ứng dụng Game demo **Dead Rising**.

Nội dung của luận văn được chia làm 4 phần; trong đó, phần 1 là chương đầu tiên giới thiệu về Game Engine và cho ta cái nhìn tổng quát về các Game Engine đang được sử dụng trên thế giới; phần 2 gồm 3 chương 2, 3, và 4 giới thiệu các thành phần và công nghệ chính sử dụng trong Nwfc Engine; phần 3 gồm 5 chương tiếp theo (từ chương 5 đến chương 9) đề cập đến các thành phần bổ sung vào Nwfc

Engine và ứng dụng Game demo Dead Rising; phần 4 là chương 10 tóm tắt kết quả đạt được và đề ra hướng phát triển trong tương lai. Cụ thể các chương như sau:

- **Chương 1 Tổng quan:** Tìm hiểu về Game Engine và 3D Engine.
- **Chương 2 Vertex Shader và Pixel Shader:** Đây là công nghệ mới nhất trong việc dựng hình 3D hiện nay và là công nghệ chính xây dựng nên Nwfc Engine.
- **Chương 3 Nwfc Engine:** Chi tiết về 3D Engine được xây dựng.
- **Chương 4 Các thuật toán Vertex và Pixel Shader:** Đề cập chi tiết đến các thuật toán chính dùng trong Vertex Shader và Pixel Shader của Nwfc Engine.
- **Chương 5 Hệ thống diễn hoạt (Animation System):** Tìm hiểu và xây dựng hệ thống quản lý các diễn hoạt cho các đối tượng trong Game.
- **Chương 6 Hệ thống vật lý (Physics System):** Tìm hiểu và ứng dụng hệ thống vật lý NovodeX vào Game.
- **Chương 7 Giới thiệu Game demo Dead Rising:** Giới thiệu về Game demo Dead Rising và các thành phần để xây dựng nên Game này.
- **Chương 8 Hệ thống hạt (Particle System) và AI:** Xây dựng hệ thống hạt và AI cho Game demo Dead Rising.
- **Chương 9 Cài đặt và hướng dẫn sử dụng:** Cài đặt, hướng dẫn sử dụng và một số kết quả nổi bật của Game demo Dead Rising.
- **Chương 10 Tổng kết:** Các kết quả đạt được và hướng phát triển.

MỤC LỤC

LỜI CẢM ƠN	i
LỜI MỞ ĐẦU	ii
MỤC LỤC	iv
DANH SÁCH CÁC HÌNH.....	viii
DANH SÁCH CÁC BẢNG.....	x
MỘT SỐ TỪ VIẾT TẮT	xi
Chương 1 Tổng quan	1
1.1. Game Engine và 3D Engine.....	2
1.2. Mối quan hệ giữa Game Engine và Game	3
1.3. Phân loại Game Engine.....	3
1.3.1. Isometric Engine	3
1.3.2. 3D FPS (First Person Shooter) Engine	4
1.3.3. MMOG (Massive Multiplayer Online Game) Engine.....	4
1.4. Một số Game Engine hiện nay	5
1.5. Tóm tắt	8
Chương 2 Vertex Shader và Pixel Shader	9
2.1. Tổng quan.....	10
2.2. Quy trình xử lý đồ họa (Graphic Pipeline)	10
2.3. Vertex Shader.....	12
2.3.1. Xử lý vertex bằng Fixed Function Pipeline	12
2.3.2. Máy ảo Vertex Shader	13
2.3.3. Cấu trúc của 1 chương trình Vertex Shader bằng hợp ngữ.....	15
2.4. Pixel Shader	17
2.4.1. Xử lý điểm ảnh bằng Fixed Function Pipeline	17
2.4.2. Máy ảo Pixel Shader	21
2.4.3. Cấu trúc của 1 chương trình Pixel Shader bằng hợp ngữ	23
2.5. Sử dụng Vertex Shader và Pixel Shader trong chương trình.....	24
2.6. Giới thiệu HLSL.....	27
2.7. Tóm tắt	28
Chương 3 Nwfc Engine.....	29
3.1. Tổng quan.....	30
3.1.1. Lý do xây dựng	30
3.1.2. Giới thiệu	30
3.2. Các tính năng của Nwfc Engine.....	31
3.3. Mô hình xây dựng Nwfc Engine.....	32
3.4. Cấu trúc của Nwfc Engine	33

3.4.1.	Các thành phần trong Nwfc module	34
3.4.2.	Các thành phần trong RendererDX9 module.....	36
3.5.	Hệ thống chất liệu (material)	37
3.5.1.	Giới thiệu	37
3.5.2.	Cấu trúc của hệ thống chất liệu (material)	38
3.5.3.	Material	39
3.5.4.	Textures.....	41
3.5.4.1.	Phân loại texture dùng trong Engine	41
3.5.4.2.	Texture flags	44
3.5.5.	Shader.....	44
3.5.5.1.	Giới thiệu tập tin Effect	46
3.5.5.2.	Định dạng tập tin Effect	46
3.5.6.	Sử dụng Vertex Shader và Pixel Shader trong Engine	48
3.5.6.1.	Vertex Shader trong Engine	49
3.5.6.2.	Pixel Shader	54
3.6.	Tóm tắt.....	54
Chương 4	Các thuật toán Vertex và Pixel Shader	55
4.1.	Lời nói đầu	56
4.2.	Đồ bóng thời gian thực Shadow Volume.....	56
4.2.1.	Cơ sở lý thuyết.....	56
4.2.2.	Vertex Shader cho Shadow Volume	62
4.2.3.	Một số kết quả đạt được.....	63
4.3.	Khung cảnh bầu trời (skybox).....	64
4.3.1.	Cơ sở lý thuyết.....	64
4.3.2.	Vertex Shader cho skybox	66
4.3.3.	Một số kết quả đạt được.....	67
4.4.	Chiếu sáng theo điểm ảnh sử dụng normal map và specular map	69
4.4.1.	Cơ sở lý thuyết.....	69
4.4.2.	Vertex Shader và Pixel Shader cho per-pixel lighting.....	75
4.4.3.	Một số kết quả đạt được.....	77
4.5.	Tóm tắt	79
Chương 5	Hệ thống diễn hoạt (Animation System).....	80
5.1.	Giới thiệu hệ thống diễn hoạt.....	81
5.2.	Các vấn đề cần giải quyết	81
5.2.1.	Tập tin lưu dữ liệu diễn hoạt.....	82
5.2.1.1.	Tập tin md5.....	82
5.2.1.2.	Xử lý dữ liệu tập tin md5.....	84
5.2.2.	Vấn đề về khung xương	85
5.2.2.1.	Giới thiệu về khung xương.....	85
5.2.2.2.	Tổ chức dữ liệu	87
5.2.2.3.	Cập nhật và di chuyển khung xương	88
5.2.3.	Đường dẫn định hướng cho diễn hoạt.....	89

5.2.3.1.	Giới thiệu về đường định hướng	89
5.2.3.2.	Cập nhật biến đổi trên các đường cơ bản	89
5.2.4.	Vấn đề về quản lý diễn hoạt.....	91
5.2.4.1.	Các vấn đề cơ bản trong diễn hoạt	91
5.2.4.2.	Tổ chức quản lý diễn hoạt	93
5.2.5.	Kết hợp các diễn hoạt.....	95
5.2.5.1.	Kết hợp các diễn hoạt khác nhau vào khung xương.....	95
5.2.5.2.	Kết hợp các diễn hoạt trong các phần của khung xương.....	96
5.3.	Hệ thống diễn hoạt trong thực thi	99
5.3.1.	Sơ đồ lớp của hệ thống diễn hoạt.....	99
5.3.2.	Chức năng các thành phần trong sơ đồ	99
5.3.2.1.	Hệ thống xử lý dữ liệu	99
5.3.2.2.	Các lớp quản lý đường dẫn.....	100
5.3.2.3.	Các lớp quản lý diễn hoạt	100
5.4.	Tóm tắt	101
Chương 6	Hệ thống vật lý (Physics System).....	102
6.1.	Giới thiệu hệ thống vật lý.....	103
6.2.	Các yếu tố cần xử lý trong hệ thống vật lý	103
6.3.	Engine vật lý NovodeX.....	104
6.4.	Sử dụng NovodeX.....	107
6.4.1.	Kết hợp NovodeX vào Game.....	107
6.4.2.	Cài đặt NovodeX trong ứng dụng.....	109
6.4.3.	Các thành phần trong sơ đồ.....	110
6.5.	Tóm tắt	114
Chương 7	Giới thiệu Game demo Dead Rising.....	115
7.1.	Giới thiệu Game demo Dead Rising	116
7.2.	Nội dung cốt truyện.....	116
7.3.	Các thành phần chính cần sử dụng.....	118
7.4.	Hệ thống các tập tin định nghĩa	118
7.4.1.	Định nghĩa giao diện (GUI).....	119
7.4.2.	Định nghĩa hệ thống hạt (Particle System)	119
7.4.3.	Định nghĩa màn chơi (Map level)	120
7.4.4.	Định nghĩa đối tượng và AI	121
7.4.5.	Các định nghĩa khác	122
7.5.	Tóm tắt.....	122
Chương 8	Hệ thống hạt (Particle System) và AI.....	123
8.1.	Hệ thống hạt (Particle System)	124
8.1.1.	Smoke particle system	124
8.1.2.	Spark particle system	125
8.1.3.	Một số hệ thống hạt được sử dụng trong Game.....	126
8.2.	Trí tuệ nhân tạo (AI)	127

8.2.1.	Cơ sở lý thuyết hành vi	127
8.2.2.	Sơ đồ trạng thái	129
8.3.	Tóm tắt	130
Chương 9	Cài đặt và hướng dẫn sử dụng.....	131
9.1.	Môi trường phát triển ứng dụng và các công cụ	132
9.2.	Kết quả đạt được	132
9.3.	Hướng dẫn sử dụng	133
9.3.1.	Các phím điều khiển	133
9.3.2.	Các chế độ chơi.....	135
9.4.	Tóm tắt	141
Chương 10	Tổng kết	142
10.1.	Kết luận	143
10.2.	Hướng phát triển và mở rộng	144
PHỤ LỤC	145
TÀI LIỆU THAM KHẢO	154

DANH SÁCH CÁC HÌNH

Hình 1-1 Vai trò của Game Engine.....	3
Hình 2-1 Sơ đồ mô tả qui trình xử lý dữ liệu và hình ảnh 3D trên Direct3D	11
Hình 2-2 Xử lý vertex bằng Fixed Function Pipeline.....	12
Hình 2-3 Sơ đồ lý thuyết máy ảo Vertex Shader	14
Hình 2-4 Cấu trúc 1 chương trình Vertex Shader bằng hợp ngữ.....	16
Hình 2-5 Qui trình xử lý đối tượng cơ sở	17
Hình 2-6 Qui trình xử lý điểm ảnh qua 2 giai đoạn	18
Hình 2-7 Mô hình xử lý điểm ảnh của Fixed Function trong giai đoạn 1	19
Hình 2-8 Mô hình xử lý điểm ảnh của Pixel Shader trong giai đoạn 1	20
Hình 2-9 Mô hình lý thuyết của máy ảo Pixel Shader.....	21
Hình 2-10 Cấu trúc chương trình Pixel Shader bằng hợp ngữ.....	23
Hình 2-11 Kết quả thực thi chương trình ví dụ về sử dụng Vertex shader.....	26
Hình 3-1 Mô hình module của Engine.....	32
Hình 3-2 Mô hình các thành phần của Nwfc Engine.....	34
Hình 3-3 Âm trả được vẽ ở chế độ khung và ở chế độ bình thường.....	38
Hình 3-4 Âm trả được vẽ với các chất liệu khác nhau.....	38
Hình 3-5 Cấu trúc của material	38
Hình 3-6 Các mặt của Environment Cube Map.....	42
Hình 3-7 Các loại texture khác nhau.....	43
Hình 3-8 Cấu trúc của 1 Shader trong Engine	45
Hình 3-9 Dựng hình nhiều lần để cho ra ảnh cuối cùng.....	45
Hình 4-1 Mô tả các phần của shadow volume	57
Hình 4-2 Cảnh bao có một mặt kẻ hướng ánh sáng còn mặt còn lại thì không	57
Hình 4-3 Dựng shadow volume mesh bằng các thêm vào các mặt phụ	59
Hình 4-4 Chương trình MeshTools tạo shadow volume mesh một cách tự động	60
Hình 4-5 Thuật toán shadow volume với kỹ thuật z-fail	61
Hình 4-6 Bối cảnh không có đổ bóng thời gian thực.....	63
Hình 4-7 Bối cảnh có đổ bóng thời gian thực.....	63
Hình 4-8 Shadow volume được vẽ bao trùm các vùng tối.....	64
Hình 4-9 Texture liền nhau ở các cạnh dùng cho sky sphere	65
Hình 4-10 Texture 6 mặt dùng cho sky box.....	65
Hình 4-11 Tọa độ của skybox được cập nhật theo tọa độ camera	66
Hình 4-12 Khung cảnh bầu trời chính diện.....	68
Hình 4-13 Một góc nhìn khác của bầu trời	68
Hình 4-14 Không gian tiếp tuyến.....	70

Hình 4-15 Tạo normal map từ height map.....	70
Hình 4-16 Tạo normal map từ vật thể có chi tiết cao hơn bằng Melody(NVidia)....	71
Hình 4-17 Chiếu sáng theo từng vertex trong Vertex Shader.....	72
Hình 4-18 Chiếu sáng trên từng điểm ảnh trong Pixel Shader	72
Hình 4-19 Sự phản xạ của tia sáng trên bề mặt	73
Hình 4-20 Tính độ phản chiếu trên từng điểm ảnh	74
Hình 4-21 Tóm tắt qui trình per-pixel lighting bằng hình vẽ	74
Hình 4-22 Các công đoạn sử dụng Fixed Function	77
Hình 4-23 Các công đoạn sử dụng Shaders per-pixel lighting	78
Hình 4-24 Kết quả sau cùng sau khi bổ sung một số hiệu ứng.....	79
Hình 5-1 Ví dụ cấu trúc khung xương	86
Hình 5-2 Ví dụ đường đi thẳng	89
Hình 5-3 Ví dụ đường đi Bezier.....	90
Hình 5-4 Ví dụ diễn hoạt qua các khung hình khóa.....	92
Hình 5-5 Sơ đồ quan hệ các lớp quản lý diễn hoạt	93
Hình 5-6 Minh họa kết hợp chuyển động các phần trong khung xương	97
Hình 5-7 Sơ đồ lớp của hệ thống diễn hoạt.....	99
Hình 6-1 Ví dụ bao bọc đối tượng Game bằng đối tượng của NovodeX	108
Hình 6-2 Các lớp chính trong hệ thống vật lý.....	109
Hình 6-3 Điều khiển nhân vật với NovodeX	113
Hình 8-1 Đặc điểm của 1 particle dạng smoke	124
Hình 8-2 Đặc điểm của 1 particle dạng spark.....	125
Hình 8-3 Một số hệ thống hạt được sử dụng trong Game	126
Hình 8-4 Các thuộc tính biểu diễn cho hành vi của quái vật	128
Hình 8-5 Sơ đồ trạng thái của quái vật	129
Hình 9-1 Màn hình giới thiệu.....	135
Hình 9-2 Màn hình tác giả	135
Hình 9-3 Màn hình chơi Game	136
Hình 9-4 Người chơi sẽ gặp nhiều quái vật trong quá trình chơi	137
Hình 9-5 Các vật thể tương tác với nhau theo đúng các định luật vật lý.....	138
Hình 9-6 Cửa tự động mở khi người chơi đến gần	138
Hình 9-7 Nhiều chi tiết được thiết kế cho khẩu súng	139
Hình 9-8 Khi bắn trúng quái vật, máu sẽ phun ra	140
Hình 9-9 Lửa bốc lên từ người quái vật.....	140
Hình 9-10 Hiệu ứng ánh sáng khi quái vật chết hay xuất hiện	141

DANH SÁCH CÁC BẢNG

Bảng 1-1 So sánh một số Game Engine.....	7
Bảng 3-1 Các cờ của texture	44
Bảng 3-2 Các hằng mặc định cơ bản.....	49
Bảng 3-3 Các tổ hợp nguồn sáng	52
Bảng 8-1 Các trạng thái của quái vật	129
Bảng 8-2 Các hành động của quái vật.....	130
Bảng 9-1 Các phím điều khiển toàn cục.....	133
Bảng 9-2 Các phím điều khiển nhân vật.....	133
Bảng 9-3 Các phím điều khiển camera ở chế độ đi theo người chơi.....	134
Bảng 9-4 Các phím điều khiển camera ở chế độ tự do	134

MỘT SỐ TỪ VIẾT TẮT

3D	3 Dimension	3 chiều
AI	Artificial Intelligence	Trí tuệ nhân tạo
ALU	Arithmetic Logic Unit	Đơn vị số học và luận lý
API	Application Program Interface	Hệ giao tiếp lập trình ứng dụng
GPU	Graphic Processor Unit	Đơn vị xử lý đồ họa
HLSL	High Level Shader Language	Ngôn ngữ shader cấp cao
PS	Pixel Shader	
VS	Vertex Shader	

Chương 1 Tổng quan

- ◇ [Game Engine và 3D Engine](#)
- ◇ [Mối quan hệ giữa Game Engine và Game](#)
- ◇ [Phân loại Game Engine](#)
- ◇ [Một số Game Engine hiện nay](#)
- ◇ [Tóm tắt](#)

1.1. Game Engine và 3D Engine

Game Engine gồm một tập hợp các thành phần khác nhau làm nền tảng tạo nên một Game (trò chơi) trên máy tính. Các thành phần cơ bản bao gồm:

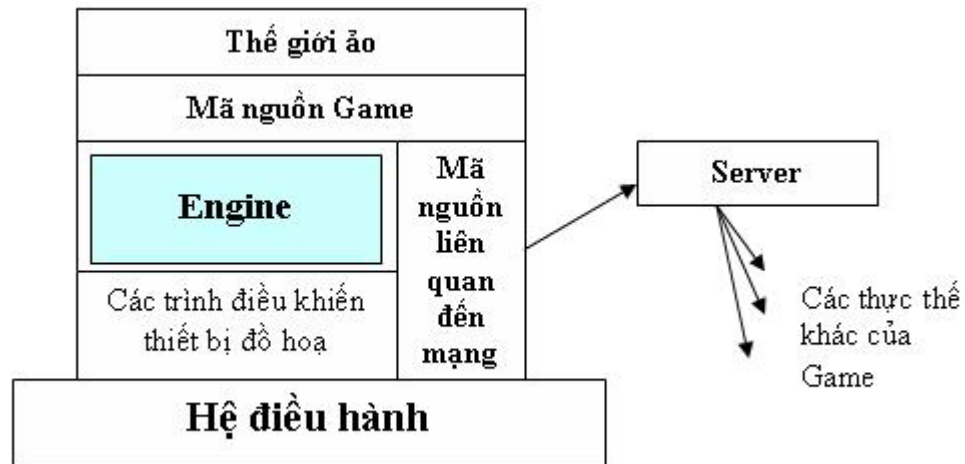
- ✚ Hệ thống toán học (Math system).
- ✚ Hệ thống xử lý tập tin dữ liệu (File system).
- ✚ Hệ thống hiển thị đồ họa (Render system).
- ✚ Hệ thống quản lý diễn hoạt (Animation system).
- ✚ Hệ thống xử lý nhập và xuất (Input and Output system).
- ✚ Hệ thống xử lý các tương tác vật lý (Physics system).
- ✚ Hệ thống xử lý trí tuệ nhân tạo (Artificial intelligence system).
- ✚ Hệ thống xử lý mạng (Network system).
- ✚ Hệ thống tạo hiệu ứng khác như khói lửa, âm thanh, ...(Effect system).

Trong các Game khác nhau thì vai trò của các thành phần trên sẽ khác nhau. Thông thường xây dựng Game ta chỉ cần một số trong các thành phần trên mà thôi. Việc xây dựng và quản lý tất cả các thành phần trên của một Game Engine đòi hỏi một lượng lớn thời gian cũng như công sức và đôi khi đòi hỏi nhiều kỹ thuật và kiến thức của nhiều lĩnh vực khác nhau.

3D Engine bao gồm một tập các hệ thống trong Game Engine nhằm giải quyết các vấn đề chính của đồ họa 3D là dựng hình 3D. Do đề tài tập trung nghiên cứu và xây dựng một 3D Engine nên chúng tôi tập trung vào xây dựng hệ thống hiển thị và hệ thống quản lý tập tin. Ngoài ra, do chúng tôi hướng đến sẽ xây dựng 3D Engine Nwfc của mình trở thành Game Engine thật sự và hiện tại cần các hệ thống khác để xây dựng Game demo hoàn chỉnh nên chúng tôi cũng đã tìm hiểu và xây dựng một số hệ thống khác của Game Engine (hiện tại chưa tích hợp vào Nwfc Engine).

Do 3D Engine là một thành phần đặc trưng của Game Engine nên chúng ta sẽ tìm hiểu về 3D Engine thông qua việc tìm hiểu về Game Engine.

1.2. Mối quan hệ giữa Game Engine và Game



Hình 1-1 Vai trò của Game Engine

Game Engine chính là thành phần cốt lõi làm nền móng xây dựng nên Game. Sự khác biệt giữa Game Engine và bản thân Game tương tự như sự khác biệt giữa động cơ và chiếc xe. Chúng ta có thể đem động cơ ra khỏi xe, chỉnh sửa và dùng lại. Một Game Engine có thể được xem như là một phần không phụ thuộc vào Game, nó cũng có thể được thay đổi độc lập với Game mà không làm thay đổi cấu trúc logic của Game.

1.3. Phân loại Game Engine

1.3.1. Isometric Engine

Đây là Game Engine dùng trong các trò chơi 2D. Các Game Engine này hiện nay có một số phần là 2D nhưng một số phần là 3D và có thể trong thời gian sắp đến sẽ chuyển hẳn sang 3D. Lý do chính của việc chuyển đổi sang 3D là để tận dụng các tính năng về ánh sáng (light) và đổ bóng (shadow) lên các đối tượng 2D.

Các Game Engine này thường được sử dụng trong các Game chiến lược (strategy) và trong các thể loại Game nhập vai (RPG: Role Game Play Genre). Một số Game được xây dựng trên Isometric Engine có thể kể đến như Baldur's Gate 2, Diablo 2 và Warlords Battle Cry 2.

Isometric Engine không phải là một Game Engine tốt cho việc xây dựng các Game chuyên về đồ họa 3D và các ứng dụng thực tại ảo (Virtual Reality) vì nó không hoàn toàn là một 3D Engine mà chỉ có một số chức năng 3D.

1.3.2. 3D FPS (First Person Shooter) Engine

Đây là Game Engine dùng nhiều nhất trong các 3D Game hiện nay. Với sự phát triển vượt bậc về công nghệ Game trong vài thập niên gần đây thì người dùng mong muốn có những Game bắt mắt nhất và loại Game Engine này đã đáp ứng được nhu cầu đó.

Một số trong rất nhiều Game được xây dựng trên các 3D FPS Engine như: Doom, Quake, Half Life, Counter Strike, Unreal, Duke Nuke'm, DeusEx, Halo, Wolfenstein, Medal of Honor, Serious Sam, Spec Ops, Dessert Storm và Hitman. 3D FPS Engine đã tạo ra những thay đổi lớn lao trong các Engine xây dựng thực tại ảo.

Các Game Engine này thông thường còn kèm theo các công cụ để tạo ra các sản phẩm mang tính nghệ thuật và tương tác cao. Các ứng dụng tạo ra các mô hình 3D nổi tiếng như 3DSMax, Maya, và Softimage đều có thể kết xuất (export) kết quả vào các Game Engine này. Các đặt trưng cơ bản của các Game Engine này là nó hỗ trợ nhiều chức năng như tô bóng điểm (pixel shader), quản lý diễn hoạt (animation), mô phỏng vật lý, mô phỏng chuyển động của các hạt nhỏ (như bụi, lửa, khói), mô phỏng chất lỏng, áo quần, và các hiệu ứng khác mà ta hiếm khi được thấy ở các 2D Engine.

1.3.3. MMOG (Massive Multiplayer Online Game) Engine

Sự khác biệt chính giữa các Game Engine đã đề cập và MMOG Engine là Game Engine này dựa trên việc lập trình trên mạng và quản lý dữ liệu thông qua mạng. Các Game xây dựng trên MMOG Engine thường chứa một cơ sở dữ liệu lớn và thực hiện trên một mạng phân tán và xử lý cho một số lượng lớn người chơi trong cùng lúc. Cũng chính vì vậy mà sự tối ưu trong việc sử dụng băng thông mạng hết

sức quan trọng và có thể nói là yếu tố sống còn của MMOG Engine. Việc nén dữ liệu và chọn lọc dữ liệu truyền trên mạng tốt có thể tiết kiệm được rất nhiều chi phí.

Hầu hết các MMOG Engine hiện nay đều tương thích và được tích hợp với một FPS Engine. Nhờ vậy ngoài các yếu tố liên quan đến mạng thì loại Engine này có các chức năng đáp ứng được các ứng dụng thực tại ảo và các yếu tố nghệ thuật được đảm bảo và đây là một yếu tố quan trọng để các người chơi hứng thú ngồi hàng giờ để chơi các Game mạng.

Một số Game được xây dựng trên MMOG Engine có thể kể đến như: Ultima Online, Ever Quest, Asheron's Call và Anarchy Online.

1.4. Một số Game Engine hiện nay

Ngày nay trên thế giới có rất nhiều Game Engine. Mỗi Game Engine được tạo ra với những mục đích, đặc tính và độ phức tạp khác nhau và do đó sẽ rất khó khăn cho người dùng trong việc chọn ra một Game Engine cho chính mình sử dụng. Hầu hết các Game Engine ngày nay cung cấp tốt các tính năng cơ bản của một Game Engine và tùy thuộc vào nhu cầu cũng như khả năng mà chúng ta chọn ra một Game Engine phù hợp cho ứng dụng của mình. Một số Game Engine tiêu biểu có thể kể đến như: Unreal - \$10,000, Quake2 - \$10,000, Quake3 - \$250,000, Torque - \$100, 3D Game Studio – \$80, Genesis - \$10,000, Lithtech - \$75,000, Crystal Space – Free, Power Render - 5,500, OpenSceneGraph, XEngine, NeoEngine, OpenApp...

Chương 1. Tổng quan

Ta có thể có một sự so sánh sơ bộ giữa các Game Engine về các chức năng mà chúng cung cấp cùng với giá tiền để có chúng qua bảng tóm tắt sau:

Game Engine	Dark Basic	Quake 1	Unreal	Halflife	Genesis	Nebula	Quake 2
Hệ thống Culling	BSP	BSP	BSP	BSP	BSP	-	BSP
Mipmap	Có	Có	Có	Có	Có	Có	Có
Map môi trường	Cubic	Có	Có	Có	Có	Có	Có
Lightmaps	Có	Có	Có	Có	Có	-	Có
Tô bóng động	Có	-	Có	-	Có	-	-
Nội suy mesh	-	-	Có	-	-	Có	Có
Terrain	Có	-	Có	-	-	Có	-
Hệ thống Particle	Có	Có	Có	Có	Có	Có	Có
Mirrors	Có	-	Có	-	Có	-	Có
Các mặt cong	-	-	Có	-	Có	-	-
Đổ bóng	Có	-	-	-	-	Có	-
Diễn hoạt khung xương	Có	-	Có	Có	Có	Có	-
Nhiều người chơi	Có	Có	Có	Có	-	Có	Có
Nhiều cảnh Game	-	-	Có	-	-	-	Có
Engine vật lý	-	-	-	-	-	-	-
Ngôn ngữ kịch bản	-	Basic	C	Basic	C	C	
Giá cả	\$100		\$10.000		\$10.000		\$10.000

(tiếp theo)

Game Engine	Game Studio	Quake 3	Lich_tech 2	Vulpine	Torque	Crystal Space	Power Render
Hệ thống Culling	BSP	BSP	Portal	Portal	BSP	BSP	Có
Mipmap	Có	Có	Có	Có	Có	Có	Có
Map môi trường	Có	Có	Có	Có	Có	Có	Có
Lightmaps	Có	Có	Có	Có	Có	Có	Có
Tô bóng động	Có	-	Có	Có	Có	Có	Có
Nội suy mesh	Có	Có	Có	Có	Có	-	Có
Terrain	Có	-	Có	Có	.	Có	.
Hệ thống Particle	Có	Có	Có	Có	Có	-	Có
Mirrors		Có	Có	Có		Có	Có
Các mặt cong	-	Có	Có	Có	-	Có	Có
Đổ bóng	-	Có	Có	Có		-	Có
Diễn hoạt khung xương	-	-	Có	Có	Có	-	Có
Nhiều người chơi	Có	Có	Có	Có	Có	-	Có
Nhiều cảnh Game	Dev	Có	Có	Có	Có	-	Có
Engine vật lý	-	-	Có	Có		-	-
Ngôn ngữ kịch bản	TLC	C++	Java	C++	C/pyth.	Python	C++
Giá cả	\$80	\$250.000	\$75.000		\$100	\$500	\$5.500

Bảng 1-1 So sánh một số Game Engine

1.5. Tóm tắt

Trong công nghệ Game tiên tiến ngày nay hầu hết các Game được xây dựng dựa trên một Game Engine. Việc xây dựng nên các Game Engine đã trở thành một xu thế tất yếu và phát triển rất mạnh mẽ. Mỗi Game Engine đều chứa đựng trong nó nhiều thành phần, tập các thành phần xử lý hiển thị 3D của Game Engine chính là 3D Engine. Chất lượng của Game Engine phụ thuộc vào chất lượng của 3D Engine, có thể nói 3D Engine chính là phần đặc trưng cơ bản nhất của Game Engine.

Chúng ta có thể chia các Game Engine ra thành 3 loại là Isometric Engine, FPS Engine và MMOG Engine. Tuy nhiên việc phân chia các Engine chỉ mang tính tương đối vì ngày nay các Engine mang trong mình rất nhiều chức năng pha trộn từ các loại khác nhằm đáp ứng việc xây dựng Game tốt nhất.

Nếu muốn xây dựng Game, ta phải tìm hiểu, so sánh các Game Engine để chọn một Game Engine phù hợp với ứng dụng và túi tiền. Việc tìm hiểu các Game Engine còn cho phép ta tạo ra một Game Engine cho chính mình để tiện sử dụng với chi phí đầu tư thấp hơn.

Chương 2 Vertex Shader và Pixel Shader

- ◇ [Tổng quan](#)
- ◇ [Quy trình xử lý đồ họa](#)
- ◇ [Kỹ thuật và lý thuyết về Vertex Shader](#)
- ◇ [Kỹ thuật và lý thuyết về Pixel Shader](#)
- ◇ [Sử dụng Vertex Shader và Pixel Shader](#)
- ◇ [Giới thiệu về HLSL](#)
- ◇ [Tóm tắt](#)

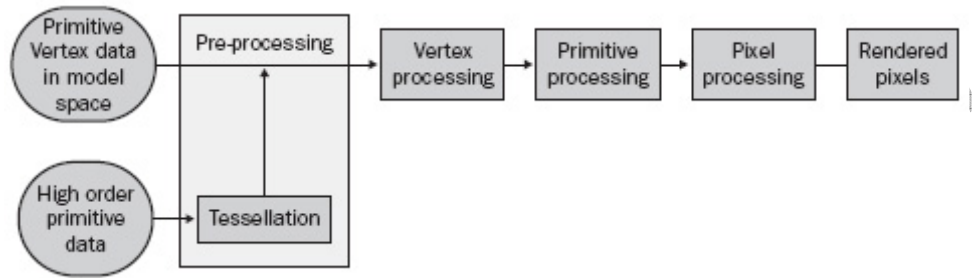
2.1. Tổng quan

Chưa có khi nào mà phần cứng dành cho đồ họa 3D phát triển như hiện nay. Phần cứng hiện nay phát triển dựa theo tiêu chí nhanh hơn, mạnh hơn, đẹp hơn. Dưới sự trợ giúp của các card đồ họa 3D tiên tiến, ranh giới giữa ảo và thực ngày càng trở nên vô cùng mong manh. Với sự ra đời của công nghệ Vertex Shader và Pixel Shader trên phần cứng, công nghiệp làm Game đã có bước tiến nhảy vọt chưa từng có, cho ra đời hàng loạt Game có chất lượng đồ họa y như thật. Vậy đâu là điều làm nên thế mạnh của công nghệ này, làm cách nào mà ta có thể ứng dụng được nó, câu hỏi đó sẽ được giải đáp trong chương này. Không đi sâu vào các khía cạnh khác, nội dung của chương này chủ yếu đề cập tới khía cạnh kỹ thuật và lý thuyết của công nghệ này.

Vì công nghệ Vertex Shader và Pixel Shader không phải là công nghệ độc lập với môi trường do có sự phụ thuộc vào 3D API được sử dụng nên trong toàn bộ báo cáo này mọi vấn đề liên quan đến Shaders đều được đề cập trên môi trường 3D API Direct3D của Microsoft (phiên bản 9.0c).

2.2. Quy trình xử lý đồ họa (Graphic Pipeline)

Công nghệ Shaders gồm 2 thành phần cơ bản là Vertex Shader (còn được gọi là Vertex Program) và Pixel Shader (hay Fragment Program) là công nghệ được tích hợp sẵn trên phần cứng cho phép người lập trình 3D hoàn toàn làm chủ quy trình xử lý dữ liệu và hình ảnh trên phần cứng (Graphic Pipeline). Trong Direct3D, Vertex Shader và Pixel Shader được gọi chung là Programmable Pipeline để có thể phân biệt với Fixed function Pipeline. Cần phải nói thêm Fixed function Pipeline là quy trình xử lý dữ liệu và hình ảnh 3D được cung cấp sẵn của Direct3D, quy trình này theo một thuật toán dựng hình cố định đối với mọi loại dữ liệu 3D đầu vào. Hình vẽ sau đây minh họa cho quy trình xử lý đồ họa (Graphic Pipeline) của Direct3D.



Hình 2-1 Sơ đồ mô tả qui trình xử lý dữ liệu và hình ảnh 3D trên Direct3D

Sơ đồ trên gói gọn toàn bộ qui trình xử lý 3D trên phần cứng của Direct3D, toàn bộ qui trình này được chia làm nhiều tầng xử lý hoàn toàn riêng biệt. Như chúng ta thấy trên sơ đồ toàn bộ qui xử lý 3D bao gồm:

✚ **Xử lý dữ liệu đỉnh (Vertex processing).** Biến đổi vertex từ không gian vật thể (model space) sang không gian chiếu (projection space).

✚ **Xử lý đối tượng cơ sở (Primitive processing).** Chuyển đổi dữ liệu vertex trong không gian chiếu thành các dữ liệu cơ sở.

✚ **Quá trình xử lý điểm ảnh (Pixel processing).** Chuyển đổi dữ liệu cơ sở thành các điểm ảnh trên màn hình (Rendered Pixels).

Trước khi đi xa hơn ta cần nắm bắt 1 số khái niệm hay thuật ngữ chuyên môn dùng trong phần này:

✚ **Fixed Function Pipeline.** Qui trình xử lý đồ họa cố định được đưa ra bởi Direct3D. Qui trình này sử dụng nhiều thuật toán 3D xử lý cố định trên các dữ liệu vào (các thuật toán này là không thể thay đổi).

✚ **Programmable Pipeline.** Qui trình xử lý đồ họa có sử dụng Vertex Shader hay Pixel Shader.

✚ **Graphic Pipeline.** Qui trình xử lý đồ họa 3D nói chung (bao gồm luôn cả Fixed Function Pipeline và Programmable Pipeline).

✚ **Vertex.** Dữ liệu đỉnh 3D. Dữ liệu trong 1 đỉnh gồm nhiều thành phần như tọa độ vị trí (position), pháp tuyến (normal), tọa độ texture (texture coordinate), màu diffuse (diffuse color), màu phản chiếu (specular color)...

✚ **Pixel.** Điểm ảnh trên màn hình

✚ **Primitive.** Đối tượng đồ họa cơ sở như tam giác, đường thẳng, hình tròn, hình vuông...

✚ **HLSL** - High Level Shader Language. Ngôn ngữ Shaders cấp cao do Microsoft phát triển tích hợp trong phiên bản Direct3D 9.0.

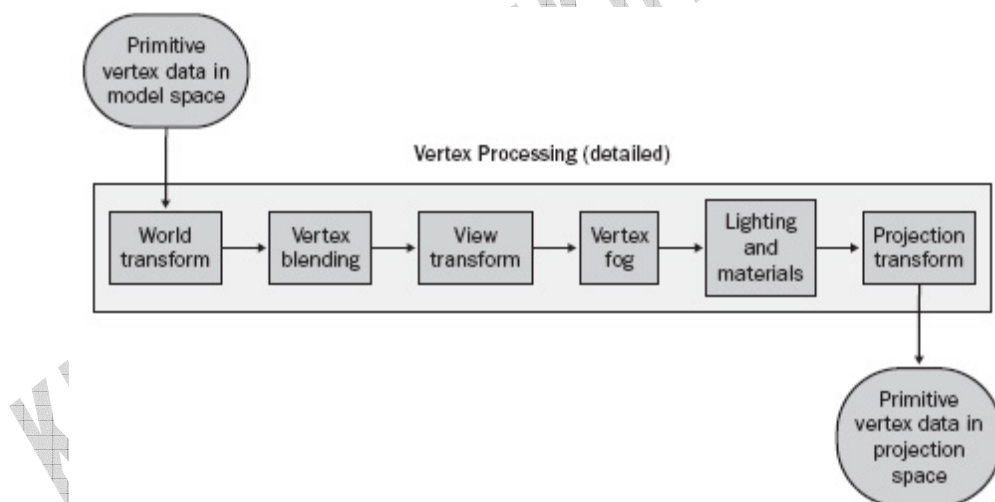
2.3. Vertex Shader

Vertex Shader là chương trình có vai trò xử lý dữ liệu vertex khi được thi hành. Vertex Shader trên Direct3D chủ yếu viết bằng hợp ngữ và HLSL (được phát triển kể từ phiên bản Direct3D 9.0). Vertex Shader là công nghệ phát triển theo các thế hệ phần cứng do đó nó có rất nhiều phiên bản khác nhau, các phiên bản cao hơn không thể chạy trên các thế hệ phần cứng cũ nhưng ngược lại thì được. Các phiên bản Vertex Shader hiện này gồm có vs_1_1, vs_2_0, vs_2_x và vs_3_0.

Vertex Shader và Fixed Function sẽ loại trừ lẫn nhau trong khi thi hành, do đó ta phải nắm được qui trình xử lý vertex của Fixed Function trước thì có thể tự viết cho mình 1 Vertex Shader như ý muốn.

2.3.1. Xử lý vertex bằng Fixed Function Pipeline

Trước khi tìm hiểu về Vertex Shader, ta hãy xem qua qui trình xử lý vertex mà Direct3D cung cấp sẵn thông qua Fixed Function Pipeline.



Hình 2-2 Xử lý vertex bằng Fixed Function Pipeline

Dữ liệu đầu vào của qui trình là dữ liệu đỉnh 3D (vertex) trong không gian vật thể (model space), trong không gian này mọi vertex đều có tọa độ tính từ gốc tọa độ của vật thể.

✚ **Biến đổi thế giới (World transform).** Biến đổi vertex từ không gian vật thể (model space) sang không gian thế giới (world space), các vertex sẽ có tọa độ tương đối với nhau trong không gian thế giới

✚ **Vertex blending.** Biến đổi tọa độ của 1 vertex trên nhiều ma trận biến đổi thế giới khác nhau. Mức độ tham gia của mỗi ma trận được xác định thông qua giá trị trọng lượng (weight) ứng với mỗi ma trận.

✚ **Biến đổi quan sát (View transform).** Biến đổi vertex từ không gian thế giới vào không gian quan sát. Vị trí của camera nằm ở gốc tọa độ của không gian quan sát, sau biến đổi này vertex sẽ có tọa độ là tọa độ tương đối đối với camera.

✚ **Tính giá trị sương mù trên từng vertex (Vertex fog).** Tính toán giá trị màu sắc của vertex khi áp dụng hiệu ứng sương mù.

✚ **Chiếu sáng (Lighting and material).** Tính toán ánh sáng trên từng vertex dựa trên mức độ phản xạ ánh sáng của vertex.

✚ **Biến đổi chiếu (Projection transform).** Biến đổi vertex từ không gian quan sát sang không gian chiếu. Đây là công đoạn cuối cùng của qui trình biến đổi.

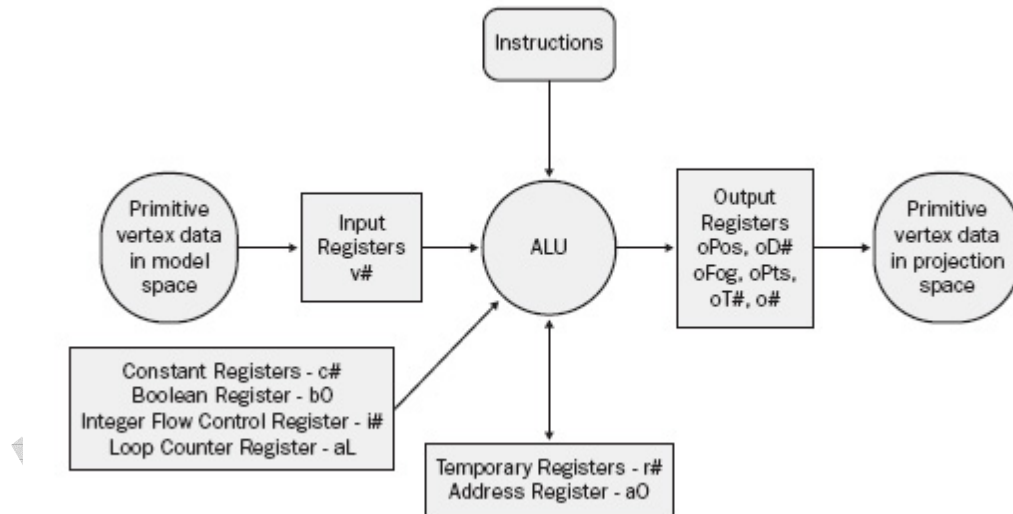
Dữ liệu đầu ra của qui trình này sẽ là đầu vào qui trình xử lý đối tượng cơ sở (Primitive processing).

Toàn bộ qui trình xử lý vertex trên của Fixed Function có thể được thay thế bằng chương trình Vertex Shader, khi đó Direct3D hoàn toàn chuyển giao quyền xử lý vertex cho Vertex Shader, Vertex Shader sau khi kết thúc sẽ trả quyền điều khiển cùng với dữ liệu xử lý được lại cho Fixed Function Pipeline để xử lý tiếp.

2.3.2. Máy ảo Vertex Shader

Để biết được cách thiết kế 1 Vertex Shader trước tiên ta cần phải nắm được mô hình máy ảo Vertex Shader (Vertex Shader Virtual Machine). Máy ảo Vertex

Shader là mô hình mang tính chất lý thuyết giúp ta dễ tiếp cận và hình dung cách thức mà Vertex Shader hoạt động. Giống như 1 loại máy móc công nghiệp, Vertex Shader nhận dữ liệu đầu vào (input), sau đó tiến hành xử lý dữ liệu đó bằng 1 số tác vụ, cuối cùng là xuất ra các thành phẩm là các dữ liệu đầu ra (output). Sau đây là sơ đồ lý thuyết:



Hình 2-3 Sơ đồ lý thuyết máy ảo Vertex Shader

Trong sơ đồ trên dữ liệu vertex được truyền từ trái sang phải. Các thanh ghi (registers) là nơi chứa và quản lý các dữ liệu đầu vào và đầu ra của Shader. Các tác vụ được thi hành trong Shader được cấu tạo từ 1 tập các vi lệnh hợp ngữ (assembly-language instructions), các vi lệnh này được thi hành ngay trên đơn vị số học và luận lý (Arithmetic Logic Unit, ALU) nằm trên GPU (Graphic Processor Unit) của card 3D. Dữ liệu đầu vào của Vertex Shader được truyền vào thông qua thanh ghi đầu vào (input registers). Vertex Shader sau khi thi hành sẽ xuất các giá trị đầu ra thông qua các thanh ghi đầu ra (output registers). Dữ liệu đầu vào của Shader là thông tin của 1 vertex được lấy từ trong vertex buffer (do chương trình cung cấp), các dữ liệu này có thể bao gồm tọa độ, pháp tuyến, tọa độ texture, màu diffuse... Dữ liệu đầu ra của Vertex Shader được trả thẳng lại cho qui trình xử lý (Graphic Pipeline) để chuyển qua công đoạn xử lý đối tượng cơ sở (Primitive processing).

Các thanh ghi được sử dụng trong Shader đều là các thanh ghi 4 chiều (có thể lưu được 4 số thực trong 1 thanh ghi). Có 4 kiểu thanh ghi, mỗi kiểu có cách sử dụng rất khác nhau.

- ✚ Thanh ghi dữ liệu vào (input registers) chứa dữ liệu đầu vào.
- ✚ Thanh ghi hằng (constant registers) chứa các hằng số dùng trong ALU.
- ✚ Thanh ghi tạm (temporary registers) chứa các dữ liệu tạm thời.
- ✚ Thanh ghi dữ liệu ra (output registers) chứa kết quả tính toán của Vertex Shader.

Với các thanh ghi đóng vai trò lưu trữ, ALU đóng vai trò thi hành các lệnh, phần quan trọng nhất của Shader chính là các vi lệnh (instructions). Vi lệnh trong Vertex Shader chủ yếu là các vi lệnh toán học thực hiện 1 tác vụ cụ thể như tính tích vô hướng (dot product), tích hữu hướng (cross product), nhân ma trận, tìm minmax... Danh sách các loại thanh ghi cũng như các vi lệnh có thể tham khảo trong Direct3D SDK.

2.3.3. Cấu trúc của 1 chương trình Vertex Shader bằng hợp ngữ

Vertex Shader nguyên thủy được xây dựng bằng hợp ngữ. Các ngôn ngữ cấp cao hơn dành cho Vertex Shader chỉ xuất hiện sau này như HLSL (chỉ có trong Direct3D 9.0 trở lên) hay GLSL (được phát triển trong phiên bản OpenGL 2.0). Phần này sẽ đề cập tới cấu trúc 1 chương trình Vertex Shader viết bằng hợp ngữ, các ngôn ngữ cấp cao sẽ được trình bày ở cuối chương này.

Một chương trình Vertex Shader viết bằng hợp ngữ căn bản được chia thành các phần sau đây:

```

vs_1_1
// constants set by the application
// c0-c3 - View+Projection matrix
// c4.x - 1
// c4.y - 0
// c4.z - 0.5
// constants assembled into the shader
def c8, 0, 1, 2, 3
def c9, 0.0f, 0.25f, 0.5f, 0.75f
...
def c24, 1,1,1,1
dcl_position v0
dcl_texcoord0 v1
// output the texture coordinates
mov oT0, v1
; Transform position
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3
    
```

Version Instruction

Comments

Constants

Input Register Declarations

Instructions

Hình 2-4 Cấu trúc 1 chương trình Vertex Shader bằng hợp ngữ

✚ **Chỉ thị phiên bản (Version Instruction).** Là thành phần đầu tiên trong chương trình, nó cho biết phiên bản Vertex Shader được biên dịch thành. Trong ví dụ trên chương trình sẽ chạy được trên phần cứng hỗ trợ vs_1_1 trở lên.

✚ **Ghi chú (Comments).** Được dùng để ghi các ghi chú trong chương trình như ý nghĩa của dữ liệu chứa trong thanh ghi... Ghi chú được bắt đầu bằng (//) hay (;) cho ghi chú 1 dòng và (/* ... */) cho ghi chú nhiều dòng.

✚ **Các hằng thanh ghi (Constants).** Các hằng được định nghĩa sau từ khoá def. Các hằng thanh ghi có thể chứa tới 4 giá trị cho mỗi thanh ghi. Như ở ví dụ trên thanh ghi c8 được gán giá trị là (0, 1, 2, 3). Các hằng thanh ghi còn có thể được gán giá trị bên trong chương trình chính thông qua phương thức *IDirect3DDevice9::SetVertexShaderConstantx*.

✚ **Định nghĩa dữ liệu trong thanh ghi đầu vào (Input Register Declarations).** Các thanh ghi dữ liệu vào như v0, v1... cần phải được định nghĩa dữ liệu trước khi sử dụng. Việc định nghĩa này sẽ giúp Direct3D ánh xạ được các dữ liệu thành phần trong vertex trên bộ nhớ vào đúng các thanh ghi tương ứng. Trong ví dụ trên, thanh ghi v0 sẽ chứa tọa độ vị trí, và v1 sẽ chứa tọa độ texture của vertex.

✚ **Các vi lệnh (Instructions).** Phần cuối cùng của 1 chương trình Vertex Shader là các vi lệnh hợp ngữ. Mọi chương trình Vertex Shader đều phải xuất giá trị ra ít nhất là vào thanh ghi vị trí oPos. Trong ví dụ trên chương trình xuất vào 2 thanh ghi là thanh ghi vị trí oPos và thanh ghi tọa độ texture oT0.

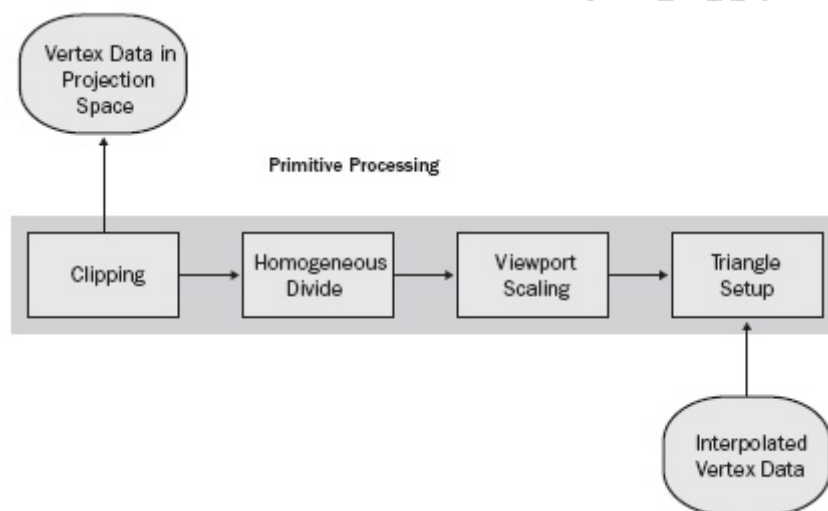
2.4. Pixel Shader

Pixel Shader là chương trình tính toán và xử lý màu trên 1 hay nhiều điểm ảnh. Pixel Shader sẽ được thực thi 1 lần cho mỗi điểm ảnh được dựng lên màn hình từ dữ liệu vertex vì thế Pixel Shader khi chạy sẽ tốn nhiều thời gian hơn Vertex Shader (chỉ xử lý 1 lần cho mỗi vertex). Pixel Shader có thể được viết bằng hợp ngữ hay HLSL. Các phiên bản hiện nay của Pixel Shader gồm có ps_1_1, ps_1_2, ps_1_3, ps_1_4, ps_2_0, ps_2_x và cuối cùng là ps_3_0.

Cũng giống như Vertex Shader, Pixel Shader khi thi hành sẽ loại trừ với Fixed Function, do đó tìm hiểu qui trình xử lý pixel của Fixed Function là điều cần thiết.

2.4.1. Xử lý điểm ảnh bằng Fixed Function Pipeline

Sau khi dữ liệu vertex được xử lý (thành tọa độ trong không gian chiếu) sẽ được chuyển qua để xử lý đối tượng cơ sở (Primitive Processing).



Hình 2-5 Qui trình xử lý đối tượng cơ sở

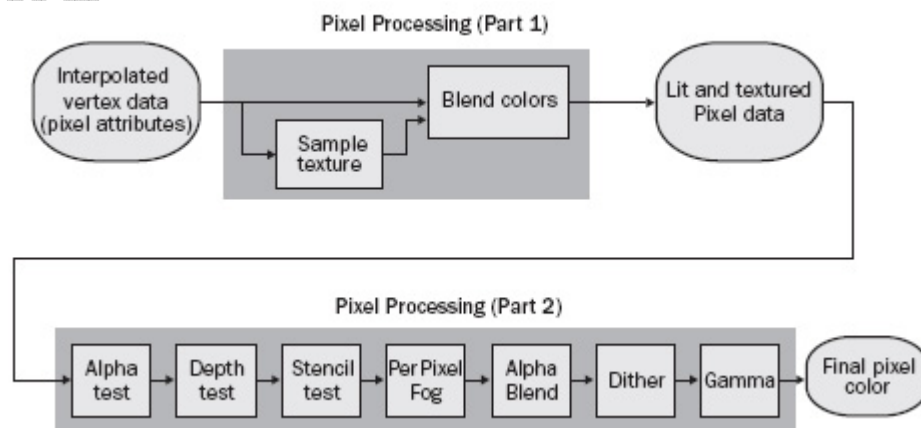
✚ **Clipping.** Loại bỏ các các đối tượng hình học không nhìn thấy được trong khối quan sát (viewing frustum) để tăng hiệu suất dựng hình.

✚ **Chuẩn hóa hệ tọa độ thuần nhất (Homogeneous Divide).** Chia các thành phần của dữ liệu cho phần tử cuối.

✚ **Ánh xạ Viewport (Viewport Scaling).** Ánh xạ dữ liệu vào tọa độ trong Viewport.

✚ **Xử lý tam giác (Triangle Setup).** Chuẩn bị cho việc nội suy tam giác và biến đổi các thuộc tính vertex thành thuộc tính điểm ảnh.

Sau khi qui trình xử lý đối tượng cơ sở hoàn tất, dữ liệu vertex đã được nội suy thành dữ liệu điểm ảnh sẵn sàng được chuyển sang qui trình xử lý điểm ảnh (điểm ảnh lúc này chưa phải là giá trị màu mà chỉ mang các thuộc tính do tính toán được từ việc nội suy tam giác mà thôi). Điểm ảnh sau đó sẽ tính toán kết hợp các thuộc tính màu sắc và lấy mẫu texture tạo thành điểm màu cuối cùng. Qui trình xử lý điểm ảnh bao gồm 2 công đoạn chính.



Hình 2-6 Qui trình xử lý điểm ảnh qua 2 giai đoạn

Giai đoạn 1 biến đổi dữ liệu nội suy trong vertex (bao gồm màu diffuse, màu specular và tọa độ texture) thành các thuộc tính màu của điểm ảnh. Gồm có các bước sau đây:

✚ **Lấy mẫu texture (Sample texture).** Lấy mẫu 1 hay nhiều texture

✚ **Hòa màu (Blend colors).** Kết hợp các màu trong thuộc tính của điểm ảnh chủ yếu là màu cơ bản (diffuse), màu phản chiếu (specular) với các màu lấy mẫu từ texture.

Giai đoạn 2 sẽ chuyển điểm màu ở cuối giai đoạn 1 thành điểm màu cuối cùng được dựng lên trên màn hình. Quá trình này bao gồm các công đoạn sau đây:

✚ **So sánh alpha (Alpha test).** Tiến hành so sánh giá trị alpha để xem màu sắc của điểm ảnh có tham gia vào giá trị màu cuối cùng hay không.

✚ **So sánh cập nhật vùng đệm độ sâu (Depth test).** Cập nhật vùng đệm độ sâu (Depth buffer) bằng độ sâu của điểm ảnh nếu điểm ảnh được vẽ.

✚ **So sánh stencil (Stencil test).** Tiến hành kiểm tra stencil nếu điểm ảnh đợi vẽ.

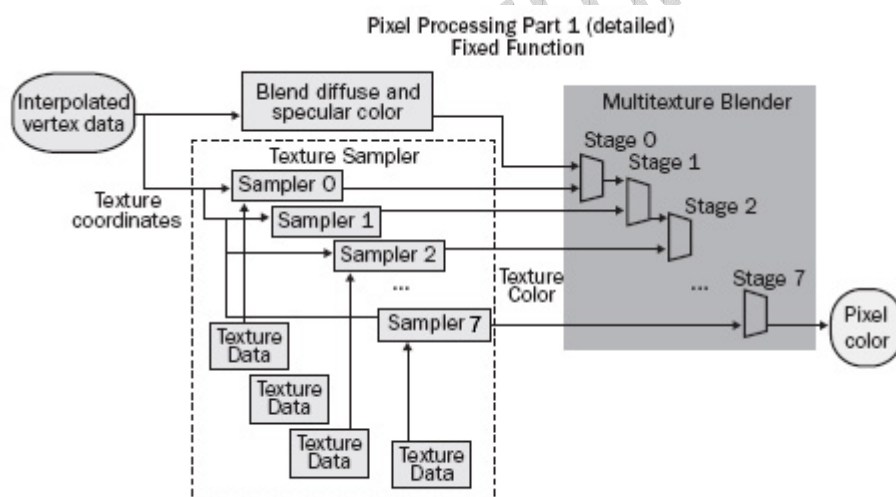
✚ **Tính toán sương mù (Per-pixel fog).** Kết hợp giá trị sương mù với màu của điểm ảnh.

✚ **Hòa màu dựa trên độ alpha (Alpha blend).** Tiến hành kết hợp màu của điểm ảnh đang vẽ với màu của điểm ảnh tương ứng trên màn hình.

✚ **Dither.** Thực hiện chuẩn hóa màu.

✚ **Hiệu chỉnh Gamma.** Thực hiện hiệu chỉnh gamma trên điểm ảnh cuối cùng.

Trong qui trình xử lý điểm ảnh ở trên, chỉ có giai đoạn 1 là có thể thay thế xử lý bằng Pixel Shader. Pixel Shader sau khi kết thúc sẽ trả giá trị màu tính được cho Fixed Function Pipeline. Ta hãy xem qua chi tiết xử lý trong giai đoạn 1 của Fixed Function Pipeline. Mô hình lý thuyết của Fixed Function Pipeline như sau:



The texture sampler is made up of texture data and samplers.

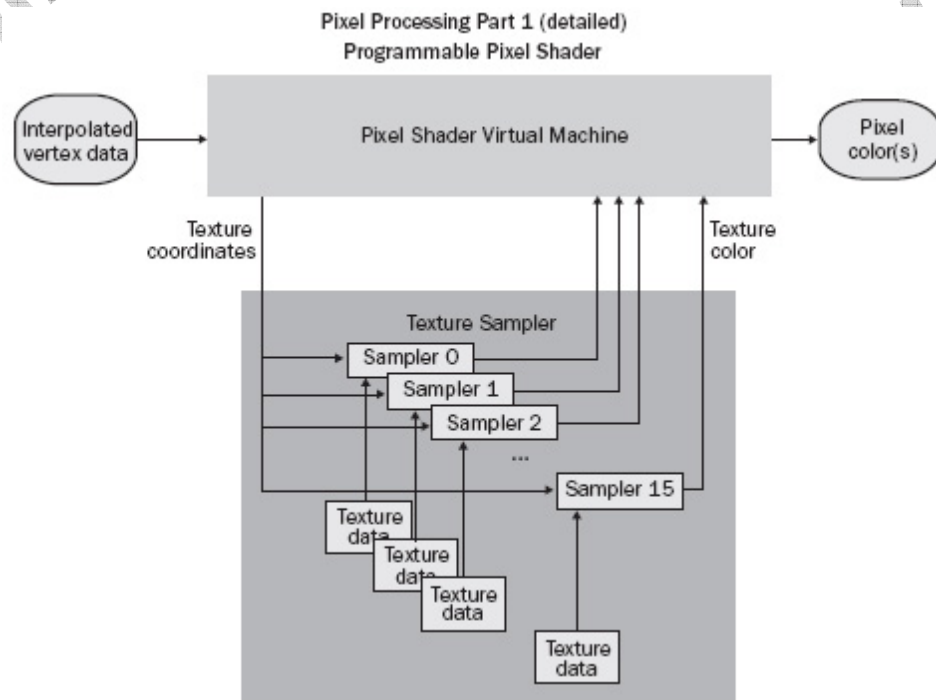
Hình 2-7 Mô hình xử lý điểm ảnh của Fixed Function trong giai đoạn 1

🚦 **Dữ liệu texture (Texture Data).** Là dữ liệu của texture lấy từ tập tin hay khung hình (Render Target).

🚦 **Sampler.** Dùng để lấy mẫu texture. Có nghĩa là dùng tọa độ texture để tìm giá trị màu tương ứng tại tọa độ đó. Các bộ lọc texture (texture filtering) có thể làm ảnh hưởng đến chất lượng mẫu nhận được (trừ chế độ lọc theo điểm (point filtering)). Fixed Function Pipeline có tất cả 8 samplers.

Cơ chế kết hợp đa texture (Multitexture Blender) bao gồm 8 tầng kết hợp (Blending Stage). Các tầng kết hợp được sắp chồng lên nhau sao cho đầu ra của đầu ra của tầng 0 trở thành đầu vào cho tầng 1, đầu ra của tầng 1 trở thành đầu vào cho tầng 2 và cứ thế. Mỗi tầng như vậy gọi là tầng texture (texture stage).

Cả giai đoạn 1 này ta có thể sử dụng Pixel Shader để xử lý thay cho Fixed Function. Mô hình lý thuyết của Pixel Shader thay thế tương ứng với Fixed Function như sau.



Hình 2-8 Mô hình xử lý điểm ảnh của Pixel Shader trong giai đoạn 1

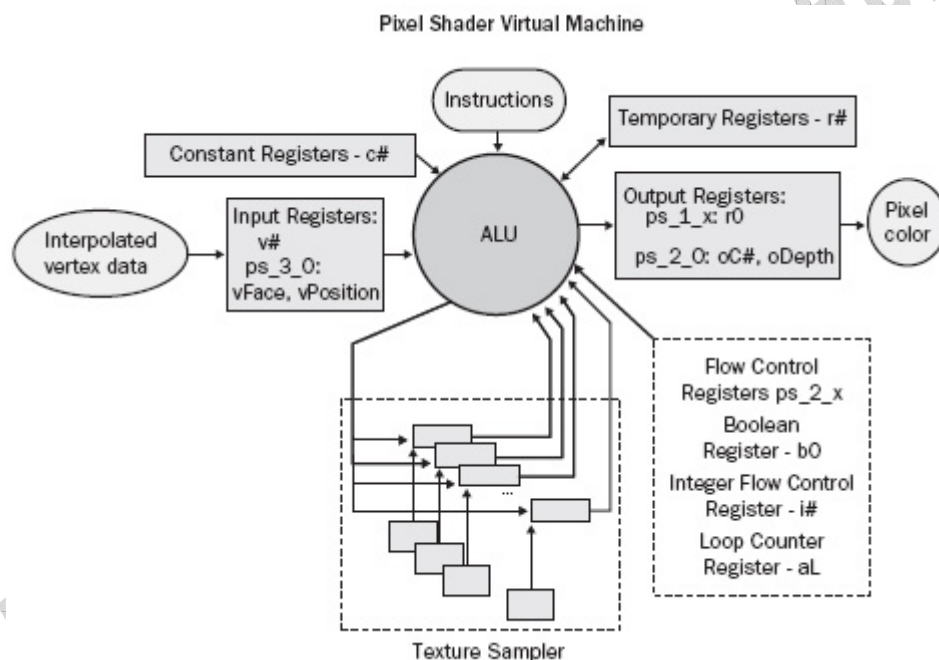
Theo mô hình trên Pixel Shader cũng dùng các samplers để lấy mẫu texture, nhưng giờ đây cơ chế kết hợp đa texture (Multitexture Blender) đã được thực hiện

ngay trong bản thân của Pixel Shader, bằng cách này các tác vụ kết hợp (blending) hoàn toàn có thể được lập trình lại. Một khi đã sử dụng Pixel Shader, ta không còn phải hiệu chỉnh các trạng thái của các tầng texture (Texture Stage States) để điều khiển cơ chế kết hợp đa texture nữa vì mọi thứ đã được làm trong Pixel Shader.

Đây chính là sức mạnh của Pixel Shader: người lập trình không còn phải tốn nhiều công sức để thiết lập các trạng thái cho các tầng texture nữa, họ giờ đây hoàn toàn có thể tự lập trình ra các thuật toán mới để sử dụng, thậm chí hoàn toàn có thể trung chuyển các luồng dữ liệu từ vertex shader vào pixel shader (nếu dùng kết hợp cả 2 shader). Tuy nhiên vẫn còn những hạn chế do người lập trình vẫn chưa can thiệp được vào giai đoạn 2 của qui trình xử lý pixel, giai đoạn này vẫn còn bị sự kiểm soát của Fixed function Pipeline. Phần sau ta sẽ xem qua kiến trúc của máy ảo Pixel Shader

2.4.2. Máy ảo Pixel Shader

Pixel Shader dùng các toán tử toán học để xử lý dữ liệu bên trong từng điểm ảnh để tính ra giá trị màu sắc cuối cùng của điểm ảnh.



Hình 2-9 Mô hình lý thuyết của máy ảo Pixel Shader

Dữ liệu trong mô hình được di chuyển từ trái sang phải. Pixel Shader sử dụng các thanh ghi (registers) để lưu trữ, quản lý các dữ liệu đầu vào (input), đầu ra (output), đồng thời thao tác trên chúng nhờ vào các vi lệnh được thực thi bởi đơn vị số học và luận lý (ALU). Pixel Shader khi thi hành sẽ biến đổi các thuộc tính bên trong của điểm ảnh (bao gồm vị trí, pháp tuyến, tọa độ texture, màu diffuse...) thành giá trị màu sắc của điểm đó. Các thanh ghi dữ liệu vào của Pixel Shader sẽ nhận dữ liệu vào là các giá trị nội suy của vertex. Chức năng của các texture sampler là dùng tọa độ texture từ các thanh ghi đầu vào để lấy mẫu texture và trả về giá trị màu lấy được.

Tương tự như Vertex Shader, Pixel Shader sẽ ghi các giá trị kết quả vào các thanh ghi đầu ra (thường là giá trị màu sắc của điểm ảnh). Thanh ghi đầu ra sau khi nhận dữ liệu sẽ trả dữ liệu về cho Graphic Pipeline để xử lý tiếp giai đoạn 2.

Sau đây là danh sách các loại thanh ghi được dùng trong Pixel Shader và chức năng của chúng.

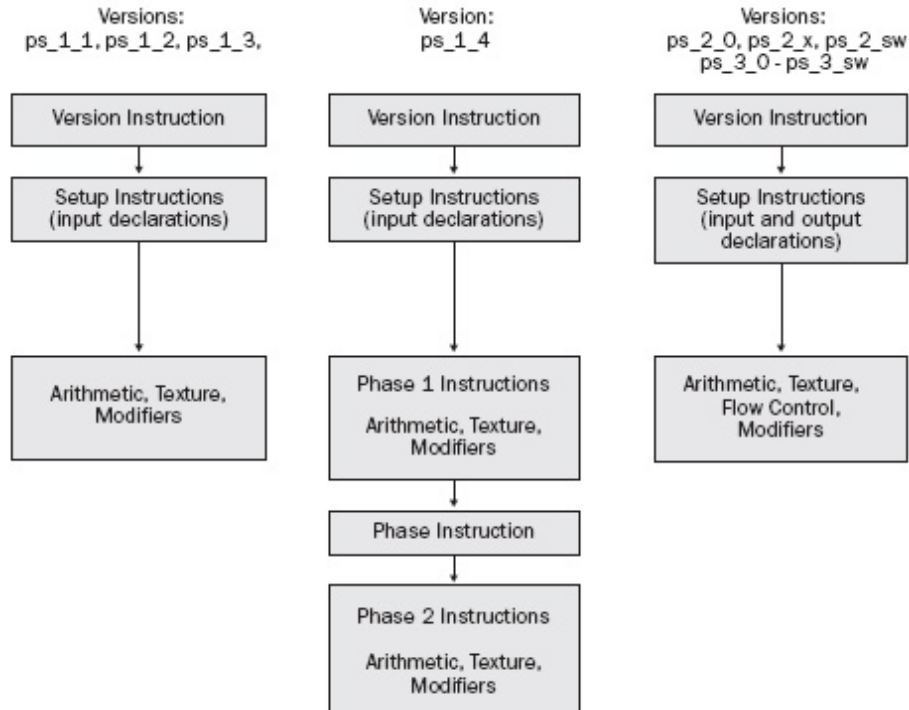
- ✚ Thanh ghi dữ liệu vào (input registers) chứa dữ liệu đầu vào (nhận được từ quá trình xử lý đối tượng cơ sở (Primitive processing)).
- ✚ Thanh ghi hằng (constant registers) chứa các hằng số dùng trong ALU.
- ✚ Thanh ghi tạm (temporary registers) chứa các dữ liệu tạm thời.
- ✚ Thanh ghi dữ liệu ra (output registers) chứa kết quả tính toán của Pixel Shader.
- ✚ Thanh ghi điều khiển (Flow control registers) điều khiển thứ tự các vi lệnh được thực thi.
- ✚ Texture sampler dùng tọa độ texture để lấy mẫu texture sau đó trả về lại cho Shader.

Các vi lệnh trong Pixel Shader chính là thành phần chính của Pixel Shader. Các vi lệnh đảm nhận việc thực thi một số tác vụ toán học trên dữ liệu ví dụ như tính tích vô hướng (dot product), nhân ma trận, tính vector đơn vị... Danh sách các loại thanh ghi cũng như các vi lệnh có thể tham khảo trong Direct3D SDK.

2.4.3. Cấu trúc của 1 chương trình Pixel Shader bằng hợp ngữ

Một chương trình Pixel Shader được cấu tạo từ nhiều dòng vi lệnh và ghi chú.

Các vi lệnh trong Pixel Shader được bố trí như sau:



Hình 2-10 Cấu trúc chương trình Pixel Shader bằng hợp ngữ

Cấu trúc chương trình Pixel Shader chia làm 3 phần chính:

✚ **Chỉ thị phiên bản (Version Instruction).** Cho biết phiên bản Pixel Shader được biên dịch thành

✚ **Các chỉ thị định nghĩa (Setup Instructions).** Định nghĩa các luồng dữ liệu trong các thanh ghi, các phiên bản sau đòi hỏi phải định nghĩa cả dữ liệu đầu vào và đầu ra.

✚ **Các vi lệnh thi hành.** Phần cuối cùng của chương trình là các vi lệnh thi hành.

Ví dụ 1 chương trình Pixel Shader

```

ps_1_1          // chỉ thị phiên bản
def c0, 0,0,0,0 // các chỉ thị định nghĩa
def c1, 1,1,1,1
def c2, 1.0,0.5,0,0
def c3, 0,-0.5,-0.25,0
tex t0          // lấy mẫu texture tại tầng 0 với tọa độ texture thứ 0
mov r0, t0      // xuất kết quả màu sắc vào thanh ghi r0
  
```

2.5. Sử dụng Vertex Shader và Pixel Shader trong chương trình

Ở các phần trước chúng ta chỉ mới tìm hiểu cách thức hoạt động và cấu trúc của 1 chương trình Vertex Shader và Pixel Shader. Nhưng Vertex Shader và Pixel Shader không phải là 1 chương trình độc lập để có thể chạy 1 cách riêng lẻ như các chương trình Window khác. Vertex Shader và Pixel Shader chỉ là các đoạn mã máy chạy trên GPU được Direct3D điều khiển trong chương trình chính. Trong phần này chúng ta sẽ xem qua 1 ví dụ cụ thể để có thể ứng dụng Vertex Shader vào trong chương trình.

Muốn dùng Vertex Shader trước tiên ta cần tạo một dự án mới có sử dụng Direct3D. Sau đó tiến hành các bước sau đây:

Công đoạn khởi tạo bao gồm:

- ✚ Khởi tạo môi trường 3D và khởi tạo các trạng thái dựng hình mặc định (Render State), tạo mới giao diện *IDirect3DDevice9*.

- ✚ Thiết kế và lập trình Vertex Shader, biên dịch Vertex Shader thành mã máy.

```
// Design a vertex shader
const char* strAsmVertexShader =
"vs_1_1          // version instruction\n"
"dcl_position v0  // define position data in register v0\n"
"m4x4 oPos, v0, c0 // transform vertices by view/proj matrix\n"
";\n"
"";
// Assemble shader
LPD3DXBUFFER pShader = NULL;
D3DXAssembleShader(
    strAsmVertexShader,
    (UINT)strlen(strAsmVertexShader),
    NULL,                // A NULL terminated array of D3DXMACROS
    NULL,                // #include handler
    D3DXSHADER_DEBUG,
    &pShader,
    NULL                 // error messages
);
```

✚ Tạo mới giao diện *IDirect3DVertexShader9* bằng phương thức *IDirect3DDevice9::CreateVertexShader*.

```
LPDIRECT3DVERTEXSHADER9 m_pAsm_VS;  
// Create the vertex shader  
g_pd3dDevice->CreateVertexShader(  
    (DWORD*)pShader->GetBufferPointer(), &m_pAsm_VS );
```

✚ Tạo mới giao diện *IDirect3DVertexDeclaration9* bằng phương thức *IDirect3DDevice9::CreateVertexDeclaration*.

```
D3DVERTEXELEMENT9 decl[] =  
{  
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,  
      D3DDECLUSAGE_POSITION, 0 }, D3DDECL_END()  
};  
// Create the vertex declaration  
g_pd3dDevice->CreateVertexDeclaration( decl, &m_pVertexDeclaration );
```

✚ Tạo mới giao diện *IDirect3DVertexBuffer9* bằng phương thức *IDirect3DDevice9::CreateVertexBuffer* và đổ dữ liệu vertex vào trong buffer, dữ liệu này sau đó sẽ được vẽ ra màn hình bằng Vertex Shader.

```
// A structure for our custom vertex type  
struct CUSTOMVERTEX  
{  
    FLOAT x, y, z; // The transformed position for the vertex  
};  
// Initialize three vertices for rendering a triangle  
CUSTOMVERTEX vertices[] =  
{  
    {-1, -1, 0}, // lower left  
    { 0, 1, 0},  // top  
    { 1, -1, 0}, // lower right  
};  
LPDIRECT3DVERTEXBUFFER9 m_pVB = 0;  
// Create the vertex buffer. Here we are allocating enough memory  
// (from the default pool) to hold three custom vertices  
g_pd3dDevice->CreateVertexBuffer(  
    3*sizeof(CUSTOMVERTEX), 0, 0, D3DPOOL_DEFAULT, &m_pVB, NULL );  
// Now we fill the vertex buffer. To do this, we need to Lock() the  
// vertex buffer to gain access to the vertices  
VOID* pVertices;  
m_pVB->Lock( 0, sizeof(vertices), (VOID**)&pVertices, 0 );  
memcpy( pVertices, vertices, sizeof(vertices) );  
m_pVB->Unlock();
```

Sau công đoạn khởi tạo ta đã có đủ các giao diện sử dụng cần thiết. Trong hàm render ta cần tiến hành các bước sau:

✚ Thiết lập các hằng cần thiết dùng trong Vertex Shader vào trong các thanh ghi hằng (constant registers) bằng các phương thức:

IDirect3DDevice9::SetVertexShaderConstantF,
IDirect3DDevice9::SetVertexShaderConstantI,
IDirect3DDevice9::SetVertexShaderConstantB

```
// Calculate World * View * Projection matrix
D3DXMATRIX compMat;
D3DXMatrixMultiply(&compMat, &m_matWorld, &m_matView);
D3DXMatrixMultiply(&compMat, &compMat, &m_matProj);
// Transpose the matrix
D3DXMatrixTranspose( &compMat, &compMat );
// Set constant
g_pd3dDevice->SetVertexShaderConstantF( 0, (float*)&compMat, 4 );
```

✚ Sử dụng *IDirect3DVertexDeclaration9* (đã tạo trước đó) bằng phương thức *IDirect3DDevice9::SetVertexDeclaration*

```
g_pd3dDevice->SetVertexDeclaration( m_pVertexDeclaration );
```

✚ Sử dụng *IDirect3DVertexShader9* (đã tạo trước đó) bằng phương thức *IDirect3DDevice9::SetVertexShader*

```
g_pd3dDevice->SetVertexShader(m_pAsm_VS);
```

✚ Vẽ dữ liệu vertex trong *IDirect3DVertexBuffer9* ra màn hình

```
g_pd3dDevice->SetStreamSource(0, m_pVB, 0, sizeof(CUSTOMVERTEX));
g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

✚ Trả quyền điều khiển về cho Fixed Function để thực hiện vẽ các đối tượng khác bằng phương thức ***IDirect3DDevice9::SetVertexShader***.

```
g_pd3dDevice->SetVertexShader(NULL);
```

Kết quả của chương trình



Hình 2-11 Kết quả thực thi chương trình ví dụ về sử dụng Vertex shader

Ý nghĩa cũng như tham số của các phương thức minh họa ở trên có thể xem trong Direct3D SDK. Ví dụ trên đây chỉ minh họa 1 chương trình Vertex Shader rất đơn giản, nhưng thực tế chương trình Vertex Shader phức tạp hơn rất nhiều, lúc đó việc lập trình bằng hợp ngữ trở nên cực kỳ khó khăn nhưng mọi việc đã được giải quyết từ khi HLSL (High-Level Shader Language) ra đời.

2.6. Giới thiệu HLSL

Trước khi DirectX 9 ra đời việc viết các Shader là một công việc thật sự nặng nhọc. Người lập trình phải tự quản lý mọi thứ từ thanh ghi cho đến các vi lệnh, họ phải tự tối ưu hóa các công việc (Vertex Shader và Pixel Shader có giới hạn 1 số lượng vi lệnh tối đa trong 1 chương trình), đó là chưa kể đến chương trình hợp ngữ rất khó đọc và kiểm lỗi. Sự ra đời các ngôn ngữ cấp cao chính là bước tiến lớn của công nghệ này giúp người lập trình dễ dàng hơn trong việc viết các Shaders.

HLSL có cấu trúc gần giống ngôn ngữ C nhưng có nhiều khác biệt do đặc thù của các chương trình Shaders. Ưu thế của chương trình viết bằng HLSL so với hợp ngữ là rất lớn vì những lý do sau:

- ✚ Chương trình dễ đọc và debug hơn.
- ✚ Lập trình dễ dàng hơn do có cấu trúc gần giống ngôn ngữ C.
- ✚ Trình biên dịch HLSL sẽ tự động tối ưu các lệnh, đồng thời tự quản lý các thanh ghi được sử dụng giúp giải phóng khá nhiều công sức của người phát triển.

Các cú pháp về ngôn ngữ này là khá nhiều và có thể xem trong Direct3D SDK. Bây giờ ta hãy xem qua 1 chương trình Vertex Shader đơn giản viết bằng HLSL.

```
float4x4 WorldViewProj;  
float4 VertexShader_Tutorial(float4 inPos : POSITION) : POSITION  
{  
    return mul(inPos, WorldViewProj);  
}
```

Trong chương trình này ta thấy có 1 khai báo biến và 1 hàm. Biến *WorldViewProj* có kiểu là 1 ma trận số thực 4x4. Ở đây ta không thấy có sự khởi tạo biến do biến này sẽ được cung cấp giá trị từ chương trình chính, đó là ma trận tổng hợp $World * View * Projection$. Ma trận này sẽ được dùng để biến đổi vertex giống như trong Fixed Function Pipeline mà phần trước đã đề cập. Hàm *VertexShader_Tutorial* có 1 giá trị trả về và 1 tham số đều có kiểu là float4 và đều

được gán ngữ nghĩa (semantic) là POSITION. Đoạn mã nguồn ở trên hoàn toàn giống ngôn ngữ C ngoại trừ kiểu dữ liệu và các ngữ nghĩa (semantic). Các kiểu dữ liệu mới được hỗ trợ chủ yếu là vector và ma trận. Ngữ nghĩa (semantic) là thành phần mới trong HLSL, ngữ nghĩa của các biến trong HLSL giúp định nghĩa loại dữ liệu sẽ được tự động truyền vào trong các biến này khi Shader được thực thi. Trong ví dụ trên dữ liệu tọa độ vị trí (POSITION) của vertex sẽ được tự động truyền vào biến inPos để chương trình xử lý, sau đó kết quả trả về sẽ được đổ lại thành ghi chứa tọa độ tương ứng trong tập các thanh ghi đầu ra.

Shaders viết bằng HLSL được sử dụng giống như các Shader viết bằng hợp ngữ ngoại trừ việc thay vì dùng hàm *D3DXAssembleShader* để biên dịch như chương trình hợp ngữ, Shaders viết bằng HLSL sẽ được biên dịch bằng hàm *D3DXCompileShader*.

2.7. Tóm tắt

Các nội dung trong chương này chủ yếu giới thiệu một cách sơ lược về công nghệ đồ họa Vertex Shader và Pixel Shader để người đọc có cái nhìn 1 cách tổng quát nhất về công nghệ này. Các chương sau sẽ đề cập tới một số thuật toán Shaders cũng như việc tích hợp công nghệ này trong 3D Engine.

Chương 3 Nwfc Engine

- ◇ [Tổng quan](#)
- ◇ [Các tính năng của Nwfc Engine](#)
- ◇ [Mô hình xây dựng Nwfc Engine](#)
- ◇ [Cấu trúc của Nwfc Engine](#)
- ◇ [Hệ thống chất liệu \(material\)](#)
- ◇ [Tóm tắt](#)

3.1. Tổng quan

3.1.1. Lý do xây dựng

Hiện nay yếu tố sống còn đối với công việc phát triển phần mềm đó chính là thời gian. Thời gian càng cao thì càng tốn nhiều chi phí để phát triển, nguy cơ thất bại hay hủy bỏ là rất lớn, ngược lại phát triển trong thời gian ngắn sẽ mang lại lợi nhuận cho người phát triển và được khách hàng tín nhiệm. Cũng giống như các phần mềm khác, Game cũng là 1 phần mềm nhưng có độ phức tạp rất lớn đòi hỏi thời gian phát triển lâu dài, nếu phát lại từ đầu thì sẽ tốn thời gian và chi phí, đó là chưa kể đến với tốc độ phát triển phần cứng như vũ bão hiện nay nếu Game phát triển quá lâu sẽ trở nên lạc hậu và có thể dẫn tới thất bại. Do đó hầu hết các nhà phát triển Game đều cố gắng dùng lại tối đa những gì có thể dùng được, nếu họ không có cái để dùng, họ sẽ mua lại của các nhà phát triển khác chính vì vậy các Game Engine đã lên ngôi. Game Engine đã chứng tỏ được sức mạnh của nó trong công việc phát triển Game, rút ngắn thời gian phát triển Game, tiết kiệm chi phí đồng thời luôn hỗ trợ các thế hệ phần cứng mới nhất. Hầu hết các Game lớn khi ra đời đều gắn mình với một Game Engine nào đó ví dụ như Game Half-life2, Doom3...

Một Game Engine tốt không chỉ có thể mở rộng và phát triển thêm mà còn phải theo kịp với sự tiến bộ của công nghệ nếu không muốn sản phẩm của mình trở nên lạc hậu. Công nghệ phần cứng phát triển ngày nay đã làm nền cho sự bùng nổ của công nghệ Shaders (Vertex Shader và Pixel Shader) trên phần cứng. Nếu điếm qua các Engine mới nhất ta đều thấy chúng đều hỗ trợ công nghệ Shaders, do đó công nghệ này cũng là đích hướng tới của các nhà phát triển Engine hiện nay.

Nhằm phát triển thử nghiệm một Engine độc lập có thể dùng lại cho nhiều ứng dụng khác hỗ trợ các thế hệ phần cứng mới nhất cũng như tích hợp công nghệ đồ họa Shaders, Nwfc Engine đã được phát triển để đáp ứng nhu cầu đó...

3.1.2. Giới thiệu

Nwfc Engine được phát triển như một thư viện độc lập có thể dùng cho phát triển Game hay các ứng dụng 3D (như 3D Editor, Level Editor, 3D Viewer...).

Đích hướng tới của Engine này là phát triển thành 1 Game Engine, tuy nhiên vì thời gian có hạn nên trong luận văn này Nwfc Engine chỉ được xây dựng số tính tăng vừa đủ nên gần giống một 3D Engine hơn là 1 Game Engine.

3.2. Các tính năng của Nwfc Engine

Engine cung cấp các tính năng cho việc phát triển dễ dàng ứng dụng 3D như Game hay các chương trình hiển thị 3D khác. Sử dụng công nghệ dựng hình đồ họa tiên tiến nhất hiện nay trên nền thư viện 3D API DirectX9.0c, mục đích của Engine là khai thác hết sức mạnh của bộ vi xử lý GPU (Graphic Processor Unit) cho việc dựng hình và hiển thị đồ họa 3D. Ngoài ra Engine đảm nhận chức năng quản lý và truy xuất hiệu quả tài nguyên phần cứng nhất là bộ nhớ Ram và card đồ họa 3D.

➤ Các tính năng chính về 3D

- ✚ Hỗ trợ tích hợp sẵn công nghệ Shaders (Vertex Shader và Pixel Shader). Cho phép viết lại các thuật toán đồ họa mới nhất mới để dùng trong ứng dụng hay có thể sử dụng các thuật toán đã được Engine cài đặt sẵn.

- ✚ Hỗ trợ đọc tập tin .X (của DirectX). Tập tin .X là một trong những format tập tin căn bản được hỗ trợ bởi Engine.

- ✚ Quản lý tự động toàn bộ các tài nguyên trên phần cứng (Ram hay card màn hình) cũng như trên bộ nhớ phụ giúp cho chương trình giảm bớt gánh nặng cho bộ nhớ.

- ✚ Hệ thống tập tin Parameter linh hoạt được sử dụng cho nhiều mục đích trong cũng như ngoài Engine. Nó cho phép người dùng tự định dạng dữ liệu riêng, chức năng gần giống như XML.

- ✚ Phần quan trọng nhất của Engine chính là hệ thống dựng hình linh hoạt dựa trên cơ sở sử dụng các chất liệu (material).

➤ Các tính năng phụ trợ

- ✚ Thư viện toán học.

- ✚ Thư viện hỗ trợ xử lý định dạng tập tin Parameter.

- ✚ Thư viện quản lý và truy xuất tập tin.
- ✚ Thư viện debug và quản lý lỗi (thư viện mã nguồn mở).
- ✚ Hệ thống giao diện lập trình dễ sử dụng và thân thiện.

3.3. Mô hình xây dựng Nwfc Engine

Nwfc Engine được triển khai dưới dạng module (mỗi module được bao bọc trong 1 DLL), gồm nhiều module liên kết lại với nhau. Tổng quát toàn bộ hệ thống Engine gồm 1 module chính và nhiều module vệ tinh. Toàn bộ Engine được thiết kế theo mô hình plug-in nên hoàn toàn có thể được phát triển mở rộng và nâng cấp.



Hình 3-1 Mô hình module của Engine

➤ **Module chính (nwfc.dll).** Đảm nhận trách nhiệm chính của toàn bộ Engine. Các trách nhiệm chính:

✚ Đây là module chính và cũng là module duy nhất giao tiếp với ứng dụng đầu cuối. Ứng dụng đầu cuối truy xuất các hàm trong module thông qua giao diện hàm (interface) mà module này cung cấp ra ngoài.

✚ Cung cấp các khai báo giao diện hàm (interface) thống nhất cho các module vệ tinh, các module vệ tinh sẽ căn cứ vào các giao diện này mà triển khai cài đặt cho phù hợp.

✚ Đảm bảo sự kết dính của các module vệ tinh với module chính hay giữa các module vệ tinh với nhau (gồm kết dính dữ liệu và kết dính hàm).

Trong Nwfc hệ thống truy xuất tập tin là duy nhất, do đó hệ thống này sẽ được chia xẻ cho toàn bộ các module vệ tinh để sử dụng. Đó là một trong các ví dụ về vai trò đảm bảo tính kết dính của module chính.

➤ **Các module vệ tinh.** Gồm nhiều module đảm nhận các chức năng khác nhau có thể hoàn toàn độc lập với nhau hay phụ thuộc lẫn nhau. Các module này có

nhiệm vụ phải hiện thực hóa các giao diện (interface) do module chính cung cấp. Ví dụ module đảm nhận chức năng dựng hình 3D bằng Direct3D, module đảm nhận chức năng truy xuất tập tin. Các module này hoàn toàn trong suốt (transparent) với ứng dụng đầu cuối, vì chúng chỉ được sử dụng nội bộ bởi module chính mà thôi. Giới thiệu sơ lược về các module sử dụng trong Engine.

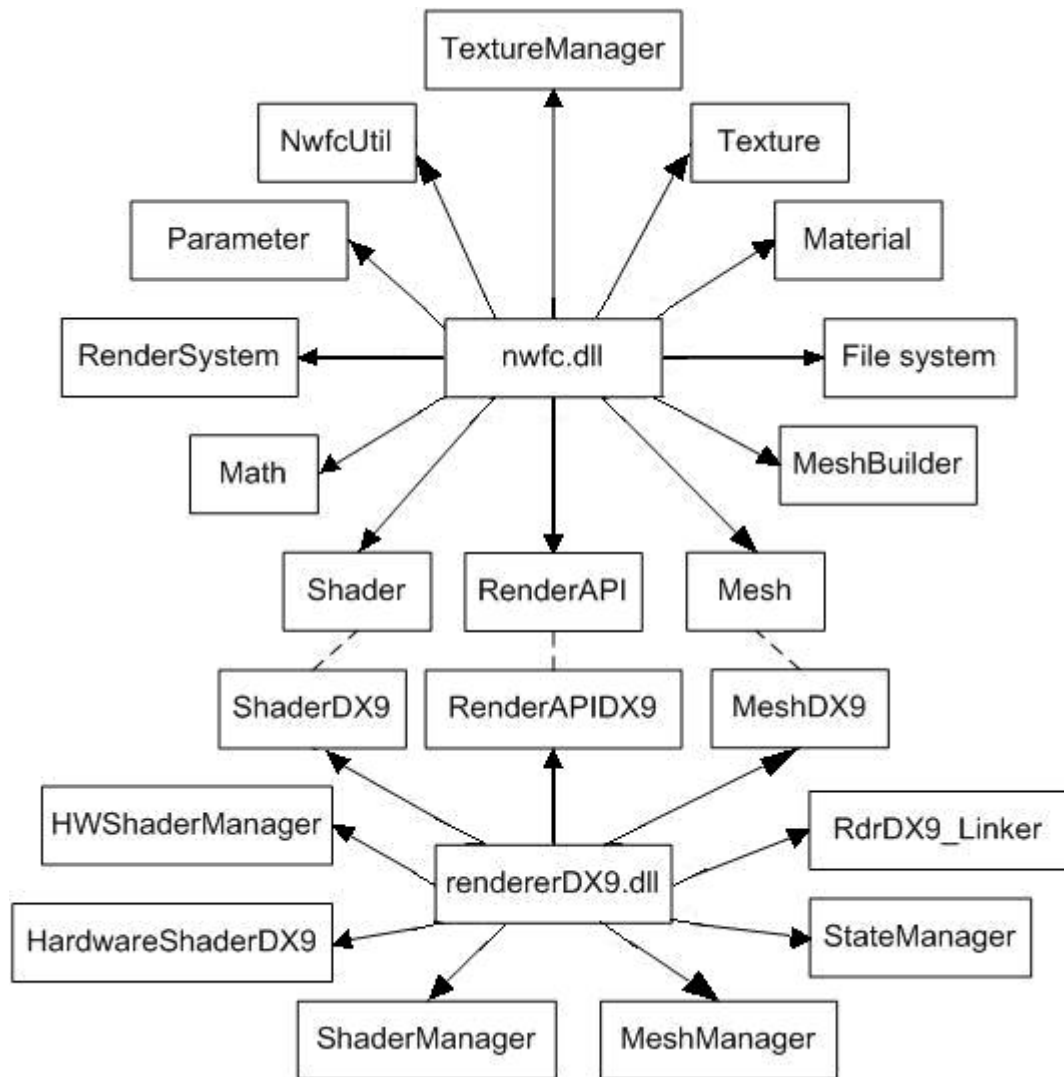
✚ Module renderer: là tập các module phụ thuộc vào thư viện đồ họa dựng hình 3D. Hiện nay trên thế giới chỉ có 2 thư viện đồ họa 3D phổ biến được nhiều người sử dụng là OpenGL và Direct3D (1 phần trong tập hợp thư viện multimedia DirectX của Microsoft), dù trong Engine chỉ được cài đặt sẵn module renderer cho thư viện Direct3D mà thôi (cụ thể là phiên bản 9.0c) nhưng ta hoàn toàn có thể thêm module dựng hình cho OpenGL bằng cách phát triển thêm module mới mà hoàn toàn không phải thông báo gì cho module chính hay compile lại code. Nhiệm vụ của module này phải hiện thực hóa các giao diện về đồ họa 3D của module chính.

✚ Module quản lý và truy xuất tập tin (file system): Đảm nhận vai trò thao tác, tìm kiếm trên tập tin cũng như đọc hay ghi tập tin, phụ thuộc vào thư viện tập tin như Standard FileIO, IOStream, hay WIN32 File System. Mọi module khác muốn truy xuất tập tin đều thông qua module này. Module này được tích hợp trong module chính mà không phải là module rời. Ta có thể tách rời module này khi ta muốn sử dụng các thư viện truy xuất tập tin khác nhau (thư viện tập tin được sử dụng trong Engine là Standard FileIO).

✚ Các module khác như module vật lý, module ngôn ngữ kịch bản (Scripting) ... sẽ được phát triển khi mở rộng Engine sau này.

3.4. Cấu trúc của Nwfc Engine

Engine là một tập các thành phần, mỗi thành phần thể hiện 1 chức năng riêng biệt. Một số chức năng chỉ sử dụng cục bộ nhưng hầu hết các chức năng được kết xuất (export) cho ứng dụng đầu cuối dưới dạng giao diện (interface).



Hình 3-2 Mô hình các thành phần của Nwfc Engine

3.4.1. Các thành phần trong Nwfc module

✚ **Mesh.** Thành phần đảm nhận chức năng lưu trữ dữ liệu 3 chiều mà chủ yếu là đỉnh (vertex) và chỉ số (index).

✚ **MeshBuilder.** Có vai trò hỗ trợ lập trình viên, giúp họ có thể sửa đổi hay thao tác trên dữ liệu 3D được chứa trong Mesh một cách nhanh và thuận tiện nhất.

✚ **Texture.** Là thành phần xử lý các ảnh bề mặt texture. Các texture là các tài nguyên chia sẻ (shared resource) và được quản lý nội bộ trong Engine bởi số đếm tham chiếu (reference count).

✚ **TextureManager.** Hệ thống các texture được quản lý nội bộ trong Engine bởi TextureManager, thành phần này sẽ đảm bảo các texture có cùng tên và đường dẫn sẽ tham chiếu đến cùng 1 đối tượng trong bộ nhớ giúp tiết kiệm rất nhiều bộ nhớ đối với các ứng dụng lớn. Các hệ thống manager và cơ chế chia xẻ tài nguyên bằng số đếm tham chiếu được sử dụng rất phổ biến trong Engine.

✚ **Material.** Đây là trái tim đồ họa của toàn bộ Engine. Thành phần này đảm nhận mọi chức năng về chất liệu hiển thị trên bề mặt 3D như độ bóng, vân bề mặt, độ phản chiếu, độ khúc xạ... Nói chung mọi vật thể đều có chất liệu của nó, gỗ thì có chất liệu gỗ, sắt thì có chất liệu kim loại. Mọi tính chất về chất liệu như thế sẽ được quản lý bởi thành phần này. Cũng giống như texture, material cũng là sử dụng reference count để quản lý chia xẻ tài nguyên.

✚ **RenderSystem.** Đây không phải là 1 thành phần mà là 1 hệ thống. Hệ thống này sẽ đảm nhận quản lý toàn bộ chức năng đồ họa của toàn bộ Engine. Mọi hành động mà ứng dụng đầu cuối muốn triển khai về mặt đồ họa đều phải thông qua hệ thống này. Hệ thống này còn đảm nhận luôn chức năng quản lý cho các material.

✚ **RenderAPI.** Đây là thành phần đóng vai trò giao tiếp với phần cứng và với các thư viện đồ họa cấp thấp giúp thực hiện các chức năng trên phần cứng. Thành phần này sẽ được hiện thực hóa trên các 3D API cụ thể (như Direct3D hay OpenGL).

✚ **Shader.** Giúp quản lý và tích hợp các thông tin cần thiết khi dựng hình như các trạng thái của phần cứng, Vertex Shader, Pixel Shader... Shader được sử dụng trong material và hoàn toàn có thể được chia xẻ giữa các material khác nhau.

✚ **FileSystem.** Hệ thống quản lý truy xuất và tìm kiếm tập tin. Mọi thành phần khác muốn thao tác trên tập tin đều phải thông qua thành phần này. Ta phải sử dụng 1 hệ thống tập tin duy nhất cho mọi thành phần vì sự phụ thuộc vào các thư viện tập tin được sử dụng như (StandardFileIO hay Win32FileIO). Ngoài ra sự quản lý tập

trung còn giúp ta triển khai các hình thức lưu trữ khác nhau (như lưu trữ trong tập tin zip chẳng hạn).

✚ **Parameter files.** Đây là thành phần hỗ trợ định dạng tập tin Parameter của Engine. Định dạng tập tin Parameter sẽ được trình bày ở phần sau. Tập tin Parameter được sử dụng rất phổ biến trong cũng như ngoài Engine. Nó giúp định nghĩa cấu trúc tập tin material... và được dùng rất nhiều cho Game demo.

✚ **Math.** Thư viện toán dùng cho 3D, hỗ trợ vector 2..4 chiều, quaternion, ma trận 4x4, color, mặt phẳng.

✚ **NwfcUtil.** Hỗ trợ ứng dụng đầu cuối có thể truy xuất các thành phần trong Engine.

3.4.2. Các thành phần trong RendererDX9 module

Các thành phần trong module này chủ yếu là các thành phần hiện thực hóa các giao diện của Nwfc module trên nền của 3D API Direct3D 9.0c. Đây là module hoàn toàn phụ thuộc Direct3D. Các thành phần:

✚ **MeshDX9.** Hiện thực hóa thành phần Mesh.


✚ **ShaderDX9.** Hiện thực hóa Shader.

✚ **RenderAPI_DX9.** Hiện thực hóa RenderAPI.

Ngoài ra module này còn nhiều thành phần mang tính chất nội bộ, chỉ được sử dụng trong module này mà thôi.


✚ **StateManager.** Quản lý trạng thái phần cứng một cách hiệu quả giúp tăng tốc độ khung hình, giảm số lần thay đổi trạng thái qua các lần gọi lệnh vẽ xuống mức thấp nhất đồng thời phục hồi lại các trạng thái đã thay đổi cho các lần vẽ sau.

✚ **HardwareShaderDX9.** Đây là thành phần chính triển khai trực tiếp công nghệ Vertex Shader và Pixel Shader trên phần cứng, do Shader là công nghệ phụ thuộc 3D API, nên thành phần phải được cài đặt trong module này.

 **HWShaderManager.** Quản lý các HardwareShaderDX9, thành phần này đảm bảo các HardwareShaderDX9 có cùng tên tập tin sẽ tham chiếu đến cùng một đối tượng.

 **MeshManager.** Quản lý các MeshDX9.

 **ShaderManager.** Quản lý các Shader.

 **RdrDX9_Link.** Đây là thành phần giúp trao đổi thông tin giữa module nwfc.dll và module renderer_DX9.dll. Module chính thông qua thành phần này sẽ truyền các thông tin cần thiết của mình vào module renderer_DX9 để module này có thể sử dụng như hệ thống tập tin (file system), các tham số cấu hình (graphic config)...

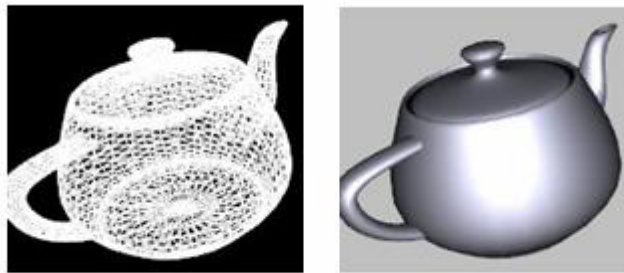
Trong các thành phần của Engine 2 thành phần là hệ thống chất liệu (material) và hệ thống tập tin parameter là 2 thành phần có tính ưu việt nhất sẽ được trình bày rõ hơn ở phần sau.

3.5. Hệ thống chất liệu (material)

3.5.1. Giới thiệu

Các vật thể trong thế giới thực đều được cấu thành bởi rất nhiều các chất liệu khác nhau. Có chất liệu thì trơn láng phản chiếu ánh sáng như bề mặt kim loại, cũng có chất liệu thì trong suốt mờ đục như các vật làm bằng plastic. Hệ thống chất liệu (material) của Nwfc Engine cũng đảm nhận vai trò gần giống như chất liệu trong thế giới thực. Material sẽ quản lý tất cả các thuộc tính làm nên bề mặt của vật thể giúp cho vật thể được hiển thị sao cho càng giống với thế giới thực càng tốt.

Nếu không có chất liệu thì đối tượng 3D khi được vẽ ra sẽ trông như thế nào ?



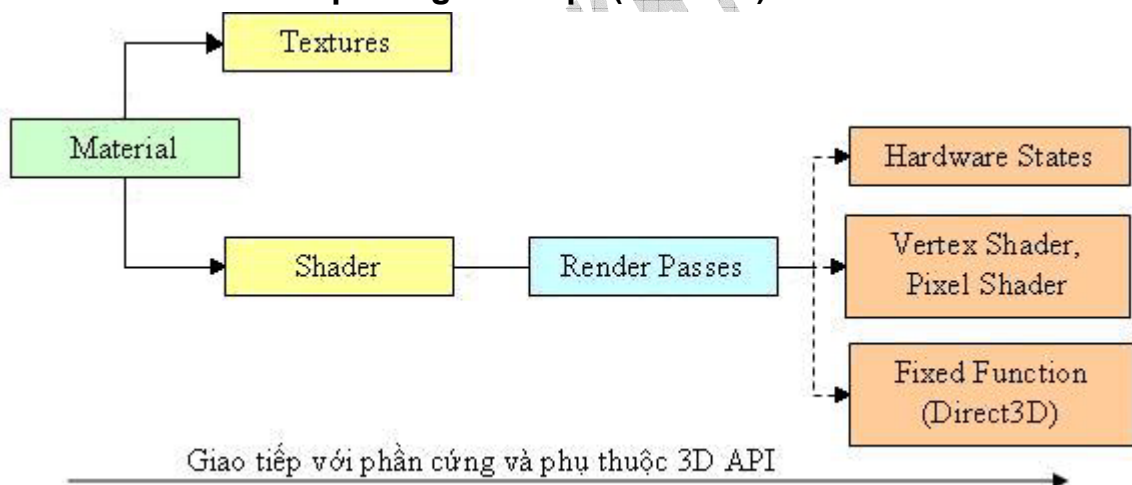
Hình 3-3 Ấm trà được vẽ ở chế độ khung và ở chế độ bình thường



Hình 3-4 Ấm trà được vẽ với các chất liệu khác nhau

Rõ ràng khi một đối tượng 3D được vẽ với các chất liệu khác nhau thì sẽ cho ra được hình ảnh rất khác nhau.

3.5.2. Cấu trúc của hệ thống chất liệu (material)



Hình 3-5 Cấu trúc của material

Cấu trúc của 1 chất liệu gồm nhiều tầng, tầng càng cao thì mức độ trừu tượng hóa càng cao và càng ít giao tiếp với phần cứng, ngược lại tầng càng thấp thì giao tiếp với phần cứng và độ phụ thuộc vào 3D API càng lớn.

✚ **Material.** Chính bản thân của chất liệu, tầng này có mức độ trừu tượng cao do ít giao tiếp với phần cứng. Material đảm nhận vai trò xử lý các thông số thuộc tính đồng thời chuyển giao các thông số này cho các tầng thấp hơn sử dụng. Các thông số này có thể được material quản lý trên tập tin hay trên bộ nhớ.

✚ **Textures.** Là các ảnh texture của bề mặt, các ảnh này có thể gồm nhiều loại khác nhau và được sử dụng cho các mục đích khác nhau.

✚ **Shader.** Là thành phần quản lý chức năng dựng hình của material, mọi chức năng vẽ của material đều phải thông qua thành phần này. Một Shader bao gồm một hay nhiều lần dựng hình (gọi là render pass).

✚ **Render passes.** Là 1 lần vẽ hình ảnh vào frame buffer. Mỗi render pass gồm các trạng thái phần cứng, vertex shader và pixel shader hay fixed function (khi không sử dụng vertex shader hay pixel shader) được sử dụng trong lần vẽ đó.

3.5.3. Material

Material trong Engine có thể tạo bằng code hay đọc từ tập tin. Định dạng tập tin của material là định dạng tập tin Parameter của Engine. Thông tin material được lưu trong tập tin bao gồm các texture và shader, material hỗ trợ tối đa 4 texture tương ứng 4 tầng texture [0..4] (texture stage) của Direct3D.

Cú pháp của 1 material như sau:

```
textures
{
    texture0      texturefile    [ texture flag(s) ]
    . . .
    texture4      texturefile    [ texture flag(s) ]
}
shader          shaderfile
```

✚ **texture[id]**. Texture tương ứng với thứ tự trong id = [0..4]. Chỉ số id trong các texture phải liên tục và không được khuyết.

✚ **texturefile**. Tên tập tin texture. Tên texture file phải bao gồm cả đường dẫn tính từ thư mục chứa tập tin thực thi đến thư mục chứa texture đó.

✚ **texture flag(s)**. 0 hay nhiều texture flag cho biết các thông tin về texture. Chi tiết các cờ này sẽ được trình bày chi tiết ở phần sau. Nếu texture flags nhiều hơn 1 thì các flag phải cách nhau khoảng trắng và toàn bộ được đặt trong ngoặc [].

✚ **shaderfile**. tên tập tin shader cần sử dụng bao gồm cả đường dẫn tính từ thư mục thực thi đến thư mục chứa tập tin đó. Ví dụ:

```
textures
{
    texture0    "textures/chair1.tga"
    texture1    "textures/chair1_local.tga"    -normalmap
    texture2    "textures/chair1_s.tga" [ -specularmap -pointsample ]
}
shader        "shaders/bump"
```

Trong trường hợp số lượng material sử dụng là rất lớn thì việc sử dụng 1 tập tin / 1 material sẽ trở nên vô cùng khó khăn vì số lượng tập tin có thể lên tới hàng trăm tập tin. Để giải quyết vấn đề đó Engine đã cung cấp khả năng tích hợp nhiều material vào trong 1 tập tin lớn gọi là material collection hay material library (matlib). Một matlib có cấu tạo như sau:

```
[material name]
{
    textures
    {
        ...
    }
    shader
}
[material name]
{
    ...
}
```

Matlib là tập hợp rất nhiều material, mỗi material trong matlib được định danh bằng tên, còn nội dung của từng material thì hoàn toàn giống như 1 material đơn thông thường.


Chức năng chính của material collection (matlib) là đơn giản hóa chức năng quản lý 1 số lượng lớn material bằng cách tích hợp rất nhiều material vào chung 1 tập tin (số lượng được tích hợp là không giới hạn), đồng thời matlib còn giúp phân loại các material thành các tập hợp giúp cho việc tìm kiếm quản lý trở nên dễ dàng.


3.5.4. Textures


Texture chính là các dữ liệu tập tin ảnh được lưu trong bộ nhớ, được sử dụng để áp vào bề mặt của vật thể trong khi render. Texture trong 3D rất đa dạng về chủng loại cũng như định dạng. Nwfc Engine hỗ trợ load các định dạng ảnh sau đây làm texture { .BMP, .DDS, .DIB, .HDR, .JPG, .PFM, .PNG, .PPM, .TGA }.

3.5.4.1. Phân loại texture dùng trong Engine

Texture trong Nwfc có thể được dùng với các mục đích sau đây.

 **Texture thường.** Dùng như tập tin ảnh thông thường, texture được sử dụng nhiều nhất trong Engine.

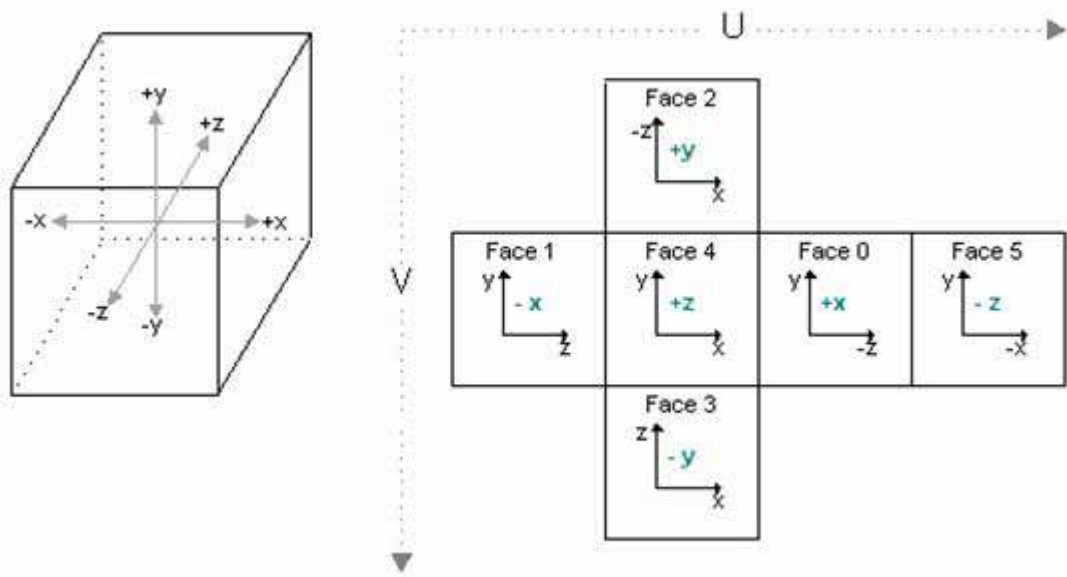
 **Normal map.** Đây là loại texture đặc biệt, thay vì chứa thông tin về màu sắc như texture thường, normal texture chứa các thông tin về không gian pháp tuyến (tangent space) của vật thể trong từng điểm ảnh. Normap map chỉ pháp huy tác dụng khi được dùng kèm với Vertex Shader và Pixel Shader, nếu không thì nó sẽ đóng vai trò như một texture thường.

 **Specular map.** Đây cũng là 1 loại texture đặc biệt, texture này chỉ chứa thông tin dạng grayscale. Texture khi áp vào mặt vật thể sẽ cho biết thông tin về mức độ phản chiếu của ánh sáng lên bề mặt vật thể (specular lighting). Cũng giống như normal map, specular map phải được dùng kèm với Vertex Shader và Pixel Shader.

Normal map và specular map thường được dùng để thực hiện thuật toán chiếu sáng trên từng điểm ảnh (per pixel lighting), per pixel lighting chỉ mới được sử dụng nhiều trong những năm gần đây, trước đó người ta vẫn sử dụng chiếu sáng

trên từng đỉnh (per vertex lighting) chủ yếu là do thiếu sự hỗ trợ từ phần cứng và công đoạn tạo ra các texture này thường tốn khá nhiều thời gian.

Environment Cube Map. Texture này rất khác với các loại trên do chứa tới 6 ảnh riêng biệt trong 1 texture. Cubemap chứa 6 ảnh mô tả khung cảnh môi trường 6 mặt xung quanh vật thể đó là các mặt Face 0, 1, 2, 3, 4, 5 tương ứng với $\{ +X, -X, +Y, -Y, +Z, -Z \}$ của khối hộp.



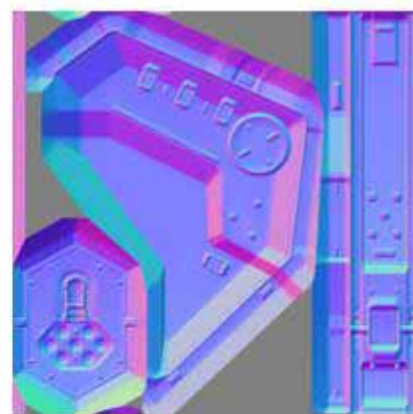
Hình 3-6 Các mặt của Environment Cube Map

Cubemap dùng chủ yếu thể hiện sự phản chiếu của môi trường xung quanh lên vật thể đối với các bề mặt phản chiếu ánh sáng như kim loại, thủy tinh...

Render Target Texture. Texture này không lấy dữ liệu từ tập tin ảnh mà dữ liệu ảnh có được là do render các vật thể vào texture. Đây là loại texture xử lý phức tạp nhất trong các loại texture nhưng ứng dụng để tạo hiệu ứng cũng nhiều nhất. Texture này được dùng để tạo các hiệu ứng cực kỳ đặc biệt như mặt nước, khúc xạ ánh sáng, motion blur..... và thường được dùng chung với Vertex Shader và Pixel Shader.



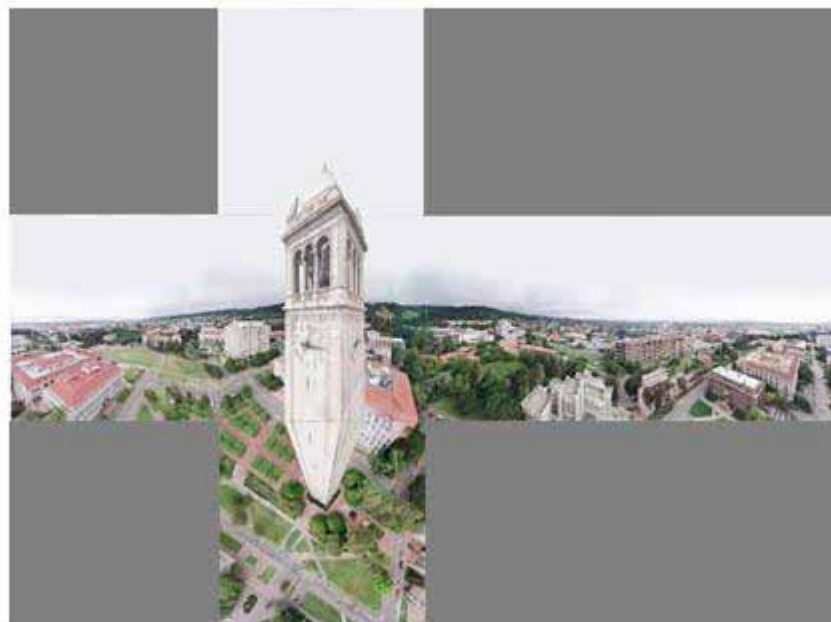
Texture thường



Normal map



Specular map



Environment Cube Map

Hình 3-7 Các loại texture khác nhau

3.5.4.2. Texture flags

Như phần trước đã đề cập, texture trong material file có đi kèm với các cờ chỉ thị. Các cờ này sẽ giúp Engine nhận dạng loại texture để sử dụng cho phù hợp, ngoài ra các cờ này còn chỉ thị cho Engine biết 1 số thuộc tính của texture.

Tên cờ	Ý nghĩa
-pointsample	Chế độ filter texture là lấy mẫu gần nhất (nearest point sample)
-nocompress	Không dùng texture nén
-clampu	Nhân bản pixel cuối khi tọa độ texture u vượt quá khoảng [0.0 .. 1.0]
-clampv	Nhân bản pixel cuối khi tọa độ texture v vượt quá khoảng [0.0 .. 1.0]
-nomipmap	Không dùng filter mipmap
-minmip	Dùng filter mipmap nhưng giới hạn mức thấp nhất là (4x4).
-onebitalpha	Cho biết texture này có 1 bit alpha
-eightbitalpha	Cho biết texture này có 8 bit alpha
-normalmap	Texture này được dùng như loại normal map
-specularmap	Texture này được dùng như loại specular map
-envcubemap	Texture này phải được load 6 mặt để dùng như Environment Cube Map
-alphaspecularmap	Texture này có thành phần alpha là specular map

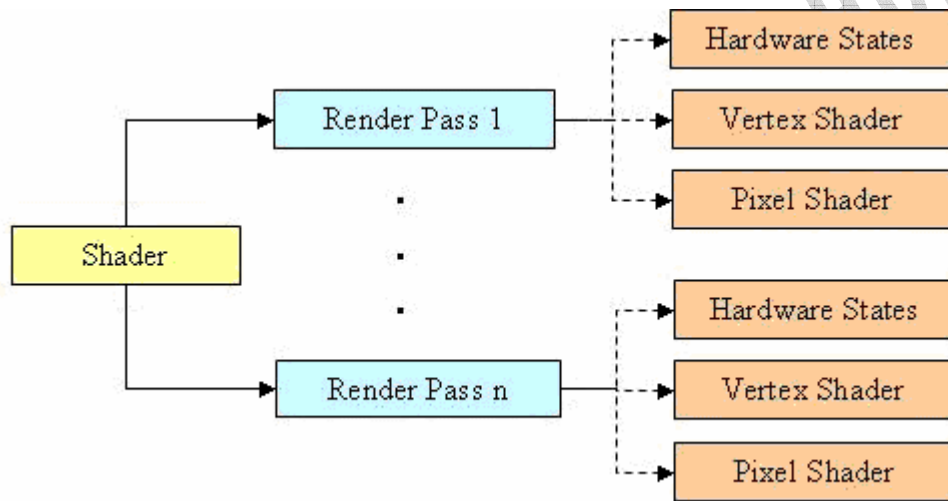
Bảng 3-1 Các cờ của texture

3.5.5. Shader

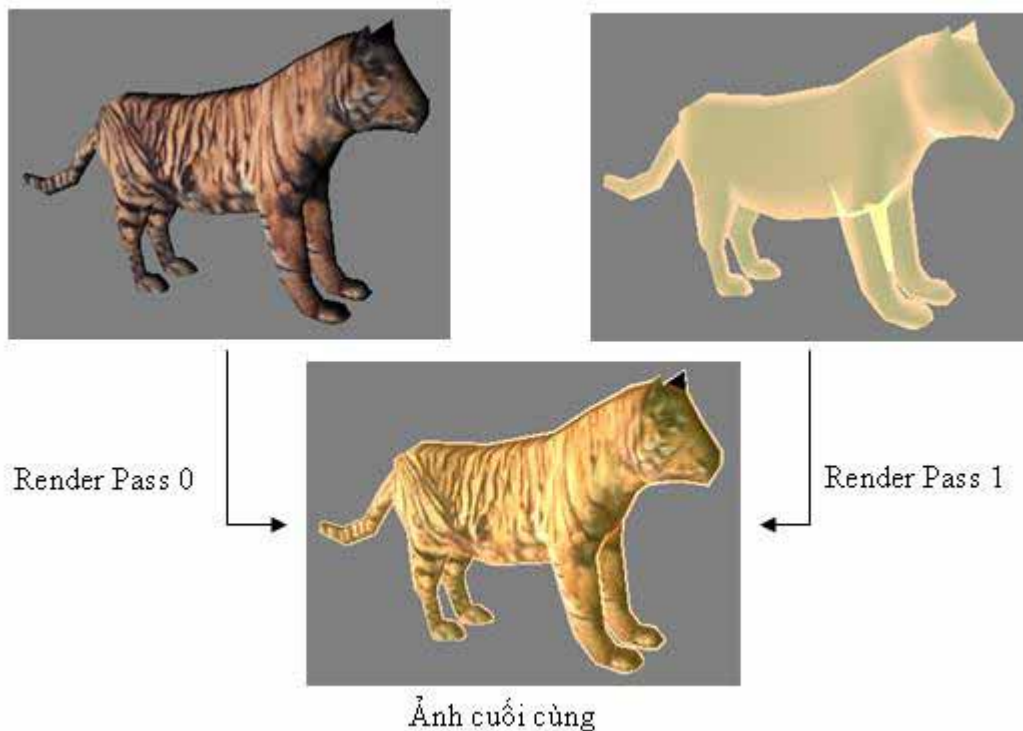
Shader là thành phần quản lý các lần dựng hình của material cũng như các thuộc tính của các lần dựng hình. Hình 3-8 thể hiện cấu trúc của 1 Shader.

Một shader bao gồm nhiều lần dựng hình (render) gọi là render pass. Mỗi render pass là một tập các trạng thái phần cứng, Vertex Shader, Pixel Shader, tuy nhiên cả

3 thành phần này không phải lúc nào cũng có mặt, ta chỉ sử dụng nó khi cần mà thôi. Sau đây là hình ảnh minh họa các render pass phối hợp với nhau.



Hình 3-8 Cấu trúc của 1 Shader trong Engine



Hình 3-9 Dựng hình nhiều lần để cho ra ảnh cuối cùng

Bởi vì 1 Shader cần rất nhiều thông tin về trạng thái phần cứng cũng như Vertex Shader và Pixel Shader nên ta cần 1 format file linh động thể có thể lưu trữ các thông tin trên, format file Shader sử dụng trong Engine là Effect file (của Direct3D).

3.5.5.1. Giới thiệu tập tin Effect

Tập tin Effect (hay FX) là định dạng tập tin đặc biệt của Direct3D. Effect file của Direct3D có thể được dùng với 3 mục đích sau đây:

- ✚ Dùng để viết Vertex Shader và Pixel Shader bằng HLSL (High Level Shader Language).
- ✚ Dùng lưu trữ các technique, là hệ thống tích hợp các render pass lại với nhau.
- ✚ Dùng tích hợp Vertex Shader, Pixel Shader với các technique.

3.5.5.2. Định dạng tập tin Effect

Một Effect file có cấu trúc bao gồm nhiều tham số (parameters), các technique, và các hàm được viết bằng ngôn ngữ HLSL. Vì Effect file rất lớn và trong Engine Effect chỉ sử dụng với mục đích như là các technique nên trong phần này chỉ đề cập tới Effect file dưới dạng các technique mà thôi.

Effect file (chỉ dùng technique) gồm nhiều technique. Mỗi technique được dùng cho một lần vẽ, nó cung cấp 1 kiểu hay cách thức dựng hình. Một technique được cấu tạo từ nhiều render pass.

➤ Cú pháp của 1 technique

```
technique [ id ] [< annotation(s) >]  
{ pass(es) }
```

- ✚ **id**. Tên của technique (có thể có hay không).
- ✚ **annotation (s)**. Gồm 0 hay nhiều nhãn của technique, nhãn trong technique được dùng để lưu trữ các thông tin riêng của người dùng.
- ✚ **pass (es)**. Gồm 0 hay nhiều pass. Mỗi pass chứa đựng nhiều trạng thái và giá trị của chúng.

➤ **Cú pháp của 1 pass:**

```
pass [ id ] [< annotation(s) >]
{ state assignment(s) }
```

✚ **id.** Là tên của pass (có thể có hay không)

✚ **annotation (s).** Gồm 0 hay nhiều nhãn của pass, nhãn trong pass được dùng để lưu trữ các thông tin riêng của người dùng.

✚ **assignment (s).** Gồm nhiều trạng thái được gán giá trị (hay các biểu thức) (danh sách các trạng thái hợp lệ có thể xem trong DirectX SDK).

Ví dụ: 1 Effect file vẽ dùng cho thuật toán shadow volume

```
technique RenderShadowVolume
{
    pass P0 <
        string vsh = "vertex_shadowvol_11";
        string psh = null;
    >
    {
        CullMode = Ccw;
        // Disable writing to the frame buffer
        AlphaBlendEnable = true;
        SrcBlend = Zero;
        DestBlend = One;
        // Disable writing to depth buffer
        ZWriteEnable = false;
        ZFunc = Less;
        // Setup stencil states
        StencilEnable = true;
        StencilRef = 1;
        StencilMask = 0xFFFFFFFF;
        StencilWriteMask = 0xFFFFFFFF;
        StencilFunc = Always;
        StencilZFail = Decr;
        StencilPass = Keep;
    }
    pass P1 <
        string vsh = "vertex_shadowvol_11";
        string psh = null;
    >
    {
        CullMode = Cw;
        StencilZFail = Incr;
    }
}
```

Trong Effect file trên chỉ có 1 technique tên là RenderShadowVolume, trong technique này sử dụng 2 pass render là P0, và P1. Trong P0 và P1 là nhiều các trạng thái được gán giá trị, các trạng thái này sau đó sẽ được Direct3D chuyển giao cho phần cứng để thiết lập. Ở đầu mỗi pass được gán nhãn là các biến kiểu string (nhãn này sẽ được xử lý nội bộ bởi hệ thống shader của Engine).

Lưu ý là các trạng thái trong pass P1 sẽ được kế thừa lại các trạng thái trong pass P0 trừ 2 trạng thái là CullMode và StencilZFail.

3.5.6. Sử dụng Vertex Shader và Pixel Shader trong Engine

Trong Effect file ví dụ ở trên thấy trong mỗi Pass đều có gán nhãn đặc biệt

```
pass P0 <
    string vsh = "vertex_shadowvol_11";
    string psh = null;
>
```

Nhãn này chính là tên của Vertex Shader và Pixel Shader được sử dụng trong pass đó. Vertex Shader được sử dụng trong khi dựng hình render pass P0 là tập tin “vertex_shadowvol_11.vsh”, và không sử dụng Pixel Shader.

Vertex Shader và Pixel Shader trong Engine được viết bằng ngôn ngữ cấp cao HLSL được biên dịch trước thành tập tin mã lệnh trong lúc xây dựng (build-time) và sử dụng trong Effect file dưới dạng tên tập tin như trong ví dụ trên.

Chương trình biên dịch được sử dụng là fxc.exe, có trong bộ DirectX SDK 9.0c. Tham số dòng lệnh của chương trình có thể xem trong SDK.

Mặc dù Effect hoàn toàn cho phép ta biên dịch Vertex Shader và Pixel Shader trong lúc thi hành (run time) thông qua 2 trạng thái là VertexShader và PixelShader (đặt trong mỗi pass) nhưng cách biên dịch trước và sử dụng dưới dạng tập tin có nhiều ưu điểm:

- + Ta có thể quản lý các tập tin này và nạp vào bộ nhớ chỉ khi cần thiết.
- + Thời gian nạp sẽ nhanh hơn do đã được biên dịch trước.

3.5.6.1. Vertex Shader trong Engine

➤ Các hằng mặc định cơ bản (constant register)

Như ta đã biết dữ liệu đầu vào của Vertex Shader gồm dữ liệu vertex và các giá trị hằng (const) được cung cấp từ chương trình thông qua các constant register. Tùy vào mục đích của thuật toán mà Vertex Shader cần các hằng khác nhau. Để làm nhẹ bớt sự quản lý, Engine cung cấp cho Vertex Shader một số lượng hằng cố định thông qua một số các constant register đã được định nghĩa sẵn. Các Vertex Shader khi được viết mới hoàn toàn có thể sử dụng các hằng này trong thuật toán của mình và cơ chế của Engine sẽ đảm bảo cho các hằng cố định này luôn có giá trị phù hợp.

Tên biến hằng	Kiểu	Register	Ý nghĩa
cEyePos	float4	c2	Điểm đặt của mắt hay camera
cModelViewProj	float4x4	c4 .. c7	Ma trận World (0) * View * Projection
cViewProj	float4x4	c8 .. c11	Ma trận View * Projection
cModelView	float4x4	c12 .. c15	Ma trận World (0) * View
cViewModel	float4x4	c17 .. c20	Ma trận World (0) * View nghịch đảo
cAmbientCube[6]	float4	c21 .. c26	Ánh sáng ambient theo 6 mặt
cLightInfo[2]	lightInfo	c27 .. c36	Thông tin về 2 nguồn ánh sáng [1]
cModel[12]	float4x3	c42 .. c77	12 ma trận World (0) -> World (12) (Dùng cho Indexed Skinning). Dùng cho phát triển Engine sau này, hiện nay Engine vẫn chưa hỗ trợ hardware skinning.
cView	float4x4	c78 .. c81	Ma trận View
cProjection	float4x4	c82 .. c85	Ma trận Projection
cInvModel	float4x4	c86 .. c89	Ma trận World (0) nghịch đảo

Bảng 3-2 Các hằng mặc định cơ bản

Chú thích:

[1] : Thông tin về nguồn sáng được bố trí thành cấu trúc.

```
struct LightInfo
{
    float4 color;
    float4 dir;
    float4 pos;
    float4 spotParams;
    float4 atten;
};
LightInfo cLightInfo[2];
```

Trong đó:

- + **color.** Màu của nguồn sáng.
- + **dir.** Hướng chiếu tới của nguồn sáng (chỉ dùng cho nguồn sáng song song (directional light)).
- + **pos.** Vị trí đặt nguồn sáng (chỉ dùng cho nguồn sáng điểm (point light) và nguồn sáng hình chóp (spot light)).
- + **spotParams.** Thông tin về nguồn sáng hình chóp (spot light).
- + **atten.** Độ suy giảm cường độ ánh sáng theo khoảng cách.

Phân loại Vertex Shader trong Engine

Vertex Shader trong Engine được phân làm 2 loại tùy vào đặc tính sử dụng

- + Vertex Shader không phụ thuộc (hay không dùng) nguồn sáng.
- + Vertex Shader phụ thuộc vào nguồn sáng

➤ Vertex Shader không phụ thuộc nguồn sáng

Vertex Shader thuộc loại này thường khá đơn giản do không phải tính toán đồ sáng từ các nguồn sáng. Số lượng các vi lệnh (intructions) thường rất ít do đó hầu hết chỉ cần dùng Vertex Shader phiên bản vs_1_1 là đủ. Các Vertex Shader được cài đặt sẵn bởi Engine trong số này gồm có (chi tiết các thuật toán và mã nguồn sẽ được trình bày ở chương sau).

- + **vertex_screenspace_11.vsh.** Dùng để vẽ các đối tượng trực tiếp lên màn hình theo tọa độ điểm trên màn hình. Thường dùng cho việc vẽ các đối tượng giao diện GUI, ngoài ra Shader này còn được dùng trong thuật toán đổ bóng Shadow Volume.

- + **vertex_shadowvol_11.vsh.** Chỉ dùng cho thuật toán đổ bóng Shadow Volume mà thôi.

- + **vertex_skybox_11.vsh.** Dùng để vẽ các khung cảnh bầu trời bằng các khối vuông.

Chỉ số 11 đằng sau tên của các Vertex Shader chính là phiên bản Vertex Shader đó đang sử dụng. Trong đó 11 là phiên bản vs_1_1, 20 là phiên bản vs_2_0, 30 là phiên bản vs_3_0.

➤ Vertex Shader phụ thuộc nguồn sáng

Các Vertex Shader trong số này gồm có:

🚦 **vertex_bump_11.vsh.** Là shader chính dùng để các đối tượng có hỗ trợ ánh sáng và bump bề mặt bằng normal map.

Vertex Shader phụ thuộc nguồn sáng có độ phức tạp hơn hẳn do phải tính toán đồ ánh sáng từ các nguồn sáng. Engine hỗ trợ tối đa 2 nguồn sáng cùng với ánh sáng môi trường 6 mặt (ambient light cube) và cung cấp thông tin các ánh sáng này thông qua các biến hằng *cAmbientCube[6]* và *cLightInfo[2]*, mỗi nguồn sáng có thể là 1 trong 3 loại sau đây, nguồn sáng càng về sau thì tính toán càng phức tạp.

🚦 Nguồn sáng song song (Directional Light)

🚦 Nguồn sáng điểm (Point Light)

🚦 Nguồn sáng hình chóp (Spot Light)

Vertex Shader phụ thuộc nguồn sáng khi được viết mới phải đảm bảo sử dụng hết các thông tin về nguồn sáng mà Engine cung cấp để việc dựng hình được chính xác.

➤ Sự phức tạp của Vertex Shader phụ thuộc nguồn sáng

Nếu không tính ánh sáng môi trường thì mỗi nguồn sáng có 4 trạng thái (không dùng, song song, điểm và hình chóp) nên tổ hợp trạng thái của 2 nguồn sáng có thể xảy ra trong Engine là $2 * 4 = 8$ (trạng thái), muốn kiểm tra 8 trạng thái này Vertex Shader phải sử dụng lệnh rẽ nhánh *if*. Ta hãy xem qua 1 Vertex Shader đơn giản chỉ tính toán màu sắc vertex theo các nguồn sáng sau đây:

```
VS_OUTPUT main( const VS_INPUT i )
{
    ...
    // Calculate lighting for light 1
    o.color = 0;
    if( cLightInfo[0].type == LIGHTTYPE_DIRECTIONAL )
        o.color += ( "do directional lighting" );
    else if ( cLightInfo[0].type == LIGHTTYPE_POINT )
        o.color += ( "do point lighting" );
}
```



```

else if ( cLightInfo[0].type == LIGHTTYPE_SPOT )
    o.color += ( "do spot lighting" );

// Calculate lighting for light 2
if( cLightInfo[1].type == LIGHTTYPE_DIRECTIONAL )
    o.color += ( "do directional lighting" );
else if ( cLightInfo[1].type == LIGHTTYPE_POINT )
    o.color += ( "do point lighting" );
else if ( cLightInfo[1].type == LIGHTTYPE_SPOT )
    o.color += ( "do spot lighting" );
...
return o;
}

```

Kết quả là chương trình này quá nặng nề và chỉ biên dịch được trên phiên bản Vertex Shader 3.0 mà thôi (do số vi lệnh phát sinh do các lệnh rẽ nhánh là rất lớn vượt quá giới hạn số vi lệnh tối đa của các phiên bản Vertex Shader thấp hơn, như phiên bản 1.1 chỉ hỗ trợ tối đa 128 vi lệnh còn 2.0 chỉ hỗ trợ 256 vi lệnh trong 1 chương trình Vertex Shader) điều đó có nghĩa là Shader này chỉ chạy được trên các card màn hình siêu cao cấp mà thôi.

➤ Cách giải quyết của Engine:

Engine chia tổ hợp các trạng thái của nguồn sáng thành 11 tổ hợp nguồn sáng (gọi là light combo) ứng với 8 trạng thái ở trên + 3 trạng thái mới do có sự tham gia của ánh sáng môi trường. Mỗi tổ hợp được gán bằng 1 chỉ số nhận dạng (từ 1..11).

Chỉ số light combo	Nguồn sáng 0	Nguồn sáng 1	Môi trường
1	NONE	NONE	NONE
2	NONE	NONE	AMBIENT
3	SPOT	NONE	AMBIENT
4	POINT	NONE	AMBIENT
5	DIRECTIONAL	NONE	AMBIENT
6	SPOT	SPOT	AMBIENT
7	SPOT	POINT	AMBIENT
8	SPOT	DIRECTIONAL	AMBIENT
9	POINT	POINT	AMBIENT
10	POINT	DIRECTIONAL	AMBIENT
11	DIRECTIONAL	DIRECTIONAL	AMBIENT

Bảng 3-3 Các tổ hợp nguồn sáng

Tại 1 thời điểm dựng hình (render) chỉ có 1 và chỉ 1 light combo tồn tại mà thôi và thông tin về các nguồn sáng của light combo này là hoàn toàn cố định. Do đó khi thiết kế Vertex Shader thay vì làm 1 Shader lớn như ở trên ta sẽ phân ra làm 11 các Shader nhỏ (mỗi Shader được định dạng bằng chỉ số ứng với light combo mà nó sử dụng, chỉ số này được gán thêm vào tên tập tin để Engine có thể nhận dạng được Vertex Shader đó được dùng cho light combo nào).

Ví dụ: tập tin “vertex_bump_11_5.vsh” trong đó “vertex_bump_11” là tên Shader + phiên bản của Shader và “_5” là chỉ số của light combo được sử dụng (ứng với tổ hợp nguồn sáng DIRECTIONAL, NONE, AMBIENT).

Với các giải quyết trên chương trình Vertex Shader không phải còn sử dụng các lệnh rẽ nhánh (if) nữa, làm cho số vi lệnh giảm xuống đáng kể khiến cho phiên bản Vertex Shader được biên dịch thành cũng giảm theo, điều này sẽ giúp chương trình có thể chạy trên nhiều thế hệ phần cứng hơn.

Đối với loại Vertex Shader phụ thuộc nguồn sáng thì cách sử dụng trong Effect file cũng có 1 số thay đổi nhỏ để Engine có thể nhận biết được loại Vertex Shader này.

```
pass p0 <
    string vsh = "vertex_bump_11?";
    string psh = "pixel_bump_20";
    >
```

Dấu “?” phía sau “vertex_bump_11” sẽ giúp Engine nhận diện đây là Vertex Shader có sử dụng nguồn sáng, Engine sẽ tự động tìm kiếm và nạp tất cả các Vertex Shader có tên “vertex_bump_11_x” (x = 1..11) vào bộ nhớ để có thể sử dụng sau này.

3.5.6.2. Pixel Shader

Hầu hết Pixel Shader trong Engine đi liền với 1 Vertex Shader tương ứng do Pixel Shader cần dữ liệu input là các output từ Vertex Shader. Pixel Shader trong Engine không sử dụng các thanh ghi hằng mặc định như Vertex Shader. Danh sách các Pixel Shader được cài đặt trong Engine.

✚ **pixel_bump_20.psh**. Là shader chính dùng để các đối tượng có hỗ trợ ánh sáng và bump bề mặt bằng normal map (dùng chung với vertex_bump_11.vsh).

✚ **pixel_glowscreen_11.psh**. Dùng để vẽ các vật thể phát sáng như bóng đèn, màn hình máy tính...

3.6. Tóm tắt

Trong chương này chúng tôi trình bày về một số thành phần chính trong Nwfc Engine. Chi tiết các thuật toán của Vertex Shader và Pixel Shader được cài đặt trong Nwfc Engine sẽ được trình bày ở chương sau.

Chương 4 Các thuật toán Vertex và Pixel Shader

- ◇ [Lời nói đầu](#)
- ◇ [Đổ bóng thời gian thực Shadow Volume](#)
- ◇ [Khung cảnh bầu trời \(sky box\)](#)
- ◇ [Chiếu sáng theo điểm ảnh \(per-pixel lighting\) sử dụng normal map và specular map](#)
- ◇ [Tóm tắt](#)

4.1. Lời nói đầu

Phần này sẽ trình bày nội dung chi tiết của thuật toán Vertex Shader và Pixel Shader dùng trong Game demo. Các kết quả thử nghiệm đều được chụp lại từ Game demo hay từ Engine.

4.2. Đồ bóng thời gian thực Shadow Volume

Trong lĩnh vực đồ họa 3D nói chung cũng như Game 3D nói riêng hiện nay, các mô hình đồ bóng thời gian thực đang được sử dụng rất rộng rãi, ngoài việc giúp người quan sát hình dung được vị trí tương đối của vật thể trong không gian 3 chiều, đồ bóng còn góp phần làm cho bối cảnh trở nên gần gũi với thực tế hơn.

Nhận thức được tầm quan trọng của việc đồ bóng thời gian thực, hàng loạt các thuật toán về đồ bóng đã đang được phát triển. Hàng loạt các thuật toán ra đời mà đi đôi với nó là chất lượng và tốc độ, trong đó có 2 thuật toán được sử dụng nhiều trong việc dựng hình 3D thời gian thực là Shadow Volume và Shadow Map. Báo cáo trong phần này sẽ đề cập tới cơ sở lý thuyết và áp dụng của thuật toán Shadow Volume trong Nwfc Engine.

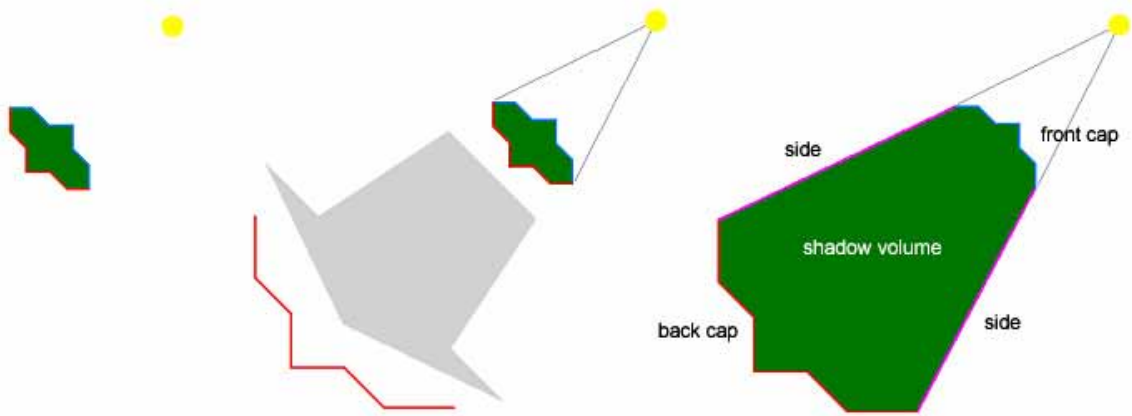
4.2.1. Cơ sở lý thuyết

➤ Vùng bóng tối (Shadow Volume)

Vùng bóng tối (shadow volume) của một vật thể là 1 khối khu vực trong không gian bị bao phủ bởi bóng tối của vật đó do một nguồn sáng phát ra. Khi dựng hình, tất cả các vật thể khác nằm trong vùng bóng tối đều không được chiếu sáng bởi nguồn sáng tạo ra vùng tối đó.

Mỗi shadow volume của vật thể được cấu tạo bởi 3 phần, phần trước (front cap), phần sau (back cap), và phần cạnh (side). Phần trước và phần sau của shadow volume được tạo bởi chính vật thể chắn sáng: phần trước được cấu tạo bởi tất cả các mặt hướng về phía ánh sáng, còn phần sau thì ngược lại bao gồm các mặt hướng ngược lại với hướng ánh sáng nhưng được di chuyển ra xa khỏi nguồn sáng theo phương ánh sáng để cấu thành vùng bóng tối, khoảng di chuyển này phải đủ lớn để

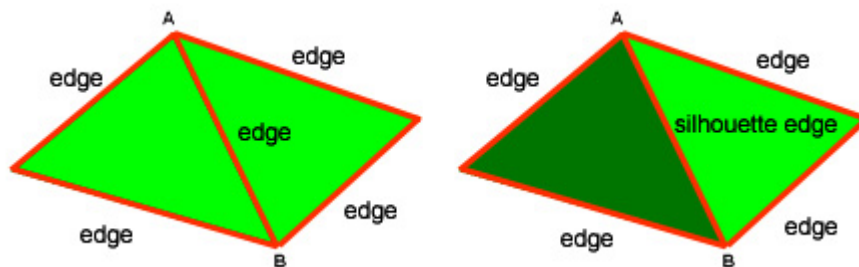
vùng bóng tối có thể bao phủ toàn bộ các vật thể khác trong bối cảnh. Phần cạnh của shadow volume được tạo ra bằng cách kéo dài (extrude) các cạnh bao (silhouette edges) theo phương chiếu của ánh sáng để tạo thành 1 vùng kín. Sau đây là hình mô tả các phần của 1 vùng bóng tối.



Hình 4-1 Mô tả các phần của shadow volume

➤ **Cạnh bao (silhouette edge)**

Điều kiện tiên quyết của thuật toán đổ bóng là ta phải tính được hình khối của shadow volume mà nội dung chính là ta phải tìm được các cạnh bao. Một cạnh (bất kỳ) được cấu tạo bởi hai điểm và có từ 1 đến 2 mặt kề liền với nó, cạnh đó được gọi là cạnh bao khi nó chỉ có 1 mặt kề hay có 2 mặt kề nhưng một mặt hướng về phía ánh sáng trong khi mặt còn lại thì không.



Hình 4-2 Cạnh bao là cạnh có một mặt kề hướng ánh sáng còn mặt còn lại thì không

Thực tế việc tìm cạnh bao đã được phát triển thành 2 thuật toán hoàn toàn riêng biệt.

✦ **Thuật toán 1:** Kiểm tra tất cả các tam giác của vật thể để tìm các cạnh có tính chất của cạnh bao. Thuật toán 1 tìm cạnh bao như sau:

Bước 1: Lập cho tất cả các tam giác của vật thể.

Bước 2: Nếu tam giác hướng về phía nguồn sáng (tích vô hướng của vector hướng ánh sáng và vector pháp tuyến của tam giác đó ≥ 0):

Bước 2-a: Chèn 3 cạnh (là 3 cặp vertices) của tam giác đó vào edge stack.

Bước 2-b: Kiểm tra trong stack xem 3 cạnh vừa chèn đó đã xuất hiện rồi hay chưa (tính luôn thứ tự đảo của cạnh, ví dụ $AB = BA$).

Bước 2-c: Nếu cạnh đó đã tồn tại trước trong stack, gỡ bỏ cả hai cạnh khỏi stack.

Bước 3: Cuối cùng, các cạnh còn lại trong stack là các cạnh bao.

Ưu điểm:

- ✚ Đơn giản do sử dụng CPU để thực hiện.
- ✚ Shadow volume tạo ra có số mặt tối thiểu, render nhanh.

Khuyết điểm:

- ✚ Tốc độ chậm do phải tính toán nhiều.
- ✚ Skinning (dùng cho diễn hoạt khung xương) phải thực hiện trước trên CPU.

✦ **Thuật toán 2:** Tạo ra một vật thể mới (shadow volume mesh) từ vật thể chắn sáng nhưng có thêm các mặt được bổ sung ở các cạnh, rồi dùng Vertex Shader để tạo hình khối của shadow volume.

Ưu điểm:

- ✚ Tốc độ nhanh do thực hiện ngay trên GPU (Vertex Shader), giải phóng CPU.
- ✚ Có thể thực hiện skinning trên phần cứng.

Khuyết điểm:

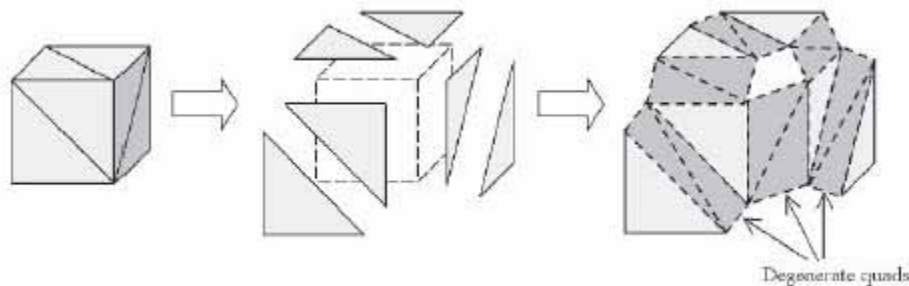
- ✚ Phức tạp do thuật toán tạo vật thể mới, việc tính toán chậm.
- ✚ Phải sử dụng thêm Vertex Shader.
- ✚ Shadow Volume có số mặt tạo ra lớn hơn rất nhiều so với vật thể gốc, render sẽ chậm hơn.

Sử dụng thuật toán 2 cho ra tốc độ nhanh hơn hẳn thuật toán 1 dù khi render có chậm hơn do có nhiều mặt hơn. Vì ưu điểm về tốc độ nên thuật toán 2 cũng là thuật toán mà Engine chọn để sử dụng nên sẽ được trình bày kỹ ở phần sau.

➤ Cách tạo Shadow Volume Mesh

Như ta đã biết nội dung chủ yếu của thuật toán 2 là phải tính được shadow volume mesh và dùng Vertex Shader để tạo hình khối shadow volume từ mesh này.

Hình vẽ sau đây minh họa cách tạo ra shadow volume mesh.



Hình 4-3 Dựng shadow volume mesh bằng các thêm vào các mặt phụ

Thuật toán tạo shadow volume mesh:

Bước 1: Lập cho tất cả các mặt trong vật thể

Bước 2: Tính vector pháp tuyến cho mỗi mặt.

Bước 3: Lập cho 3 cạnh của mỗi mặt.

Bước 3-a: Thêm cạnh đó vào 1 list kiểm tra.

Bước 3-b: Nếu cạnh đó đã xuất hiện ở trong list (ta đã tìm thấy cạnh được dùng chung cho 2 mặt):

- + Nếu pháp tuyến của các mặt kề cạnh đó không song song với nhau, thêm 1 tứ giác (degenerate quad) vào list kết quả.

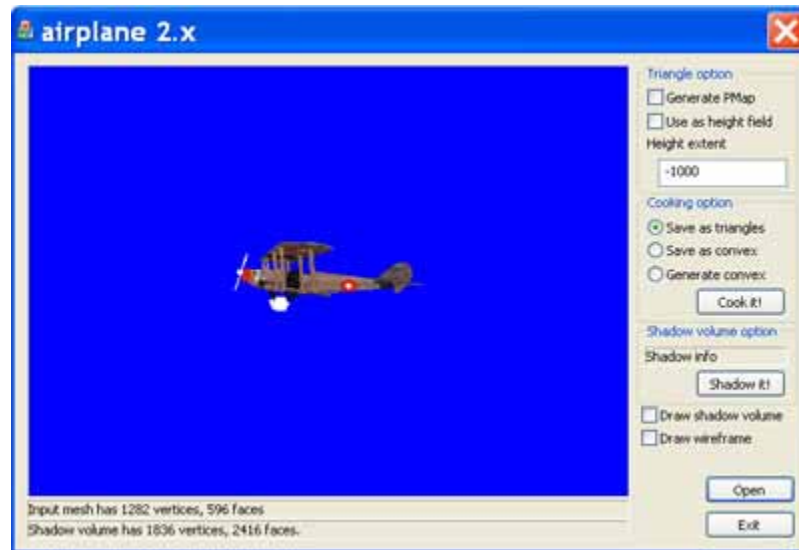
- + Ngược lại, chỉ thêm cạnh đó vào list kết quả.

Bước 3-c: Gỡ bỏ cạnh đang xử lý và các cạnh tương tự ra khỏi list kiểm tra.

Bước 4: Tạo mảng dữ liệu để chứa dữ liệu của shadow volume mesh, mỗi vertex của shadow volume mesh chỉ gồm vị trí và pháp tuyến mà thôi.

Bước 5: Nếu còn cạnh nào trong list kiểm tra thì vật thể đang xử lý không phải là khối đặc vì trong khối đặc tất cả các cạnh đều có 2 mặt kề với nó.

Trong chương trình Game demo việc tạo shadow volume mesh đã được tự động hóa bằng chương trình MeshTools được phát triển kèm theo Game (cách sử dụng chương trình này xem thêm ở phần phụ lục). MeshTools nhận đầu vào là vật thể gốc sau đó tạo shadow volume mesh và lưu vào .X file để load vào Game sau này.



Hình 4-4 Chương trình MeshTools tạo shadow volume mesh một cách tự động

➤ Dựng hình bóng tối (render shadow)

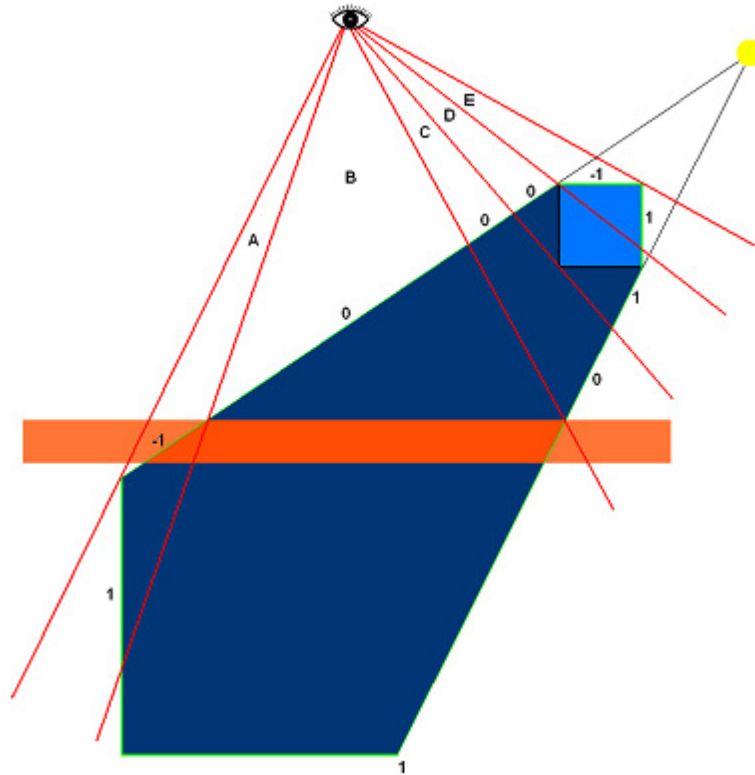
Sau khi tính được hình khối shadow volume ta phải vẽ hình khối này để tạo thành các vùng bóng tối trong bối cảnh. Ý tưởng chủ đạo của thuật toán này giống như cách tìm 1 điểm trong hình khối. Ta kẻ 1 đoạn thẳng từ mắt tới điểm cần xét, nếu đoạn thẳng đó chỉ đi vào hình khối shadow volume mà không có đi ra (tức là cắt shadow volume 1 số lẻ lần) thì điểm cần xét nằm trong vùng tối.

Để đếm số lần cắt cho mỗi điểm ảnh được xét người sử dụng vùng đệm stencil buffer để lưu số lần cắt qua các shadow volume. Stencil buffer là vùng đệm bộ nhớ bổ sung (thường được chia sẻ chung với vùng đệm độ sâu (depth buffer)), vai trò chủ yếu của vùng đệm này là làm mặt nạ (mask) cho các pixel được vẽ.

Qua quá trình phát triển có 2 thuật toán được sử dụng cho bài toán dựng hình bóng tối là z-pass và z-fail. Mỗi thuật toán có ưu khuyết điểm riêng nhưng trong khuôn khổ bài báo cáo này chủ yếu sẽ trình bày về thuật toán z-fail. Chi tiết của thuật toán này như sau:

- + Vẽ các mặt sau (back face) của shadow volume. Nếu độ sâu của điểm ảnh so sánh thất bại (thường là giá trị lớn hơn giá trị trong depth buffer), giá trị của stencil buffer tại điểm đó sẽ tăng lên 1.
- + Vẽ các mặt trước (front face) của shadow volume. Nếu độ sâu của điểm ảnh so sánh thất bại, giá trị của stencil buffer giảm đi 1.

Sau khi vẽ shadow volume bằng thuật toán trên tất cả các điểm trong bối cảnh bị phủ bởi bóng tối có giá trị trong stencil buffer khác 0 trong khi các điểm khác thì bằng 0. Sau đây là hình vẽ minh họa cho thuật toán dựng bóng tối (z-fail)



Hình 4-5 Thuật toán shadow volume với kỹ thuật z-fail

Trong hình vẽ trên vật màu cam biểu diễn cho vật thể nhận bóng tối, vật màu xanh là vật thể chắn sáng. Các khu vực A, B, C, D, E là 5 khu vực sẽ được dựng hình mà có sự ảnh hưởng của shadow volume. Các giá trị ở các vùng là giá trị trong stencil buffer thay đổi khi mặt trước (front face) hay mặt sau (back face) của shadow volume được vẽ vào frame hình. Tại khu vực A và E, mặt trước và sau khi vẽ đều khiến giá trị trong stencil buffer không đổi do trong 2 khu vực này vật nhận bóng tối (màu cam) và vật chắn sáng (màu xanh) gần mắt hơn nên làm cho việc kiểm tra độ sâu thất bại khiến giá trị trong stencil buffer trung hòa về 0. Trong khu vực B và D, mặt trước sẽ vượt qua sự kiểm tra độ sâu trong khi mặt sau thì thất bại, vì thế giá trị stencil tại các vùng này sẽ mang giá trị 1 (do chỉ có mặt sau làm stencil buffer thay đổi mà thôi). Ở khu vực C, cả mặt trước và sau đều vượt qua sự kiểm tra độ sâu, nên không làm cho giá trị trong stencil thay đổi. Khi kết thúc quá trình vẽ shadow volume thì stencil ở khu vực B, D khác 0, cho thấy B và D nằm trong vùng bóng tối của vật chắn sáng.

Cuối cùng ta chỉ cần phủ tối vùng B, D bằng cách vẽ một tứ giác lớn bao phủ toàn bộ bối cảnh là kết thúc thuật toán.

4.2.2. Vertex Shader cho Shadow Volume

Ở các phần trên ta đã nắm được cơ sở lý thuyết của thuật toán này. Phần này sẽ trình bày Vertex Shader được dùng để vẽ shadow volume.

```
static const int g_lightIndex = 0;
static const float g_extrudeDistance = 200.0f;
static const float g_depthEpsilon = 1e-5f;
static const float3 g_shadowColor = { 1.0f, 1.0f, 0.0f };
struct VS_INPUT {
    float4 position : POSITION;
    float3 normal : NORMAL;
};
struct VS_OUTPUT {
    float4 position : POSITION;
    float4 color : COLOR0;
};
VS_OUTPUT main( VS_INPUT i )
{
    VS_OUTPUT o;
    // Calculate vertex world position
    float3 worldPos = mul( i.position, cModel[0] );
    // Calculate vertex world normal
    float3 worldNormal = mul( i.normal, cModel[0] );
    // Calculate light-to-vertex vector in world space
    float3 lightVector = normalize( worldPos -
        cLightInfo[g_lightIndex].pos );
    // Extrude if the vertex not facing the light
    if( dot(worldNormal, -lightVector) < 0.0 )
        worldPos += lightVector * g_extrudeDistance;
    // Calculate projection space position
    float4 projPos = mul( float4(worldPos, 1), cViewProj );
    // Offset an amount to avoid z-fighting
    projPos.z += g_depthEpsilon * projPos.w;
    o.position = projPos;
    // Final color
    o.color = float4( g_shadowColor, 0.1f );
    return o;
}
```

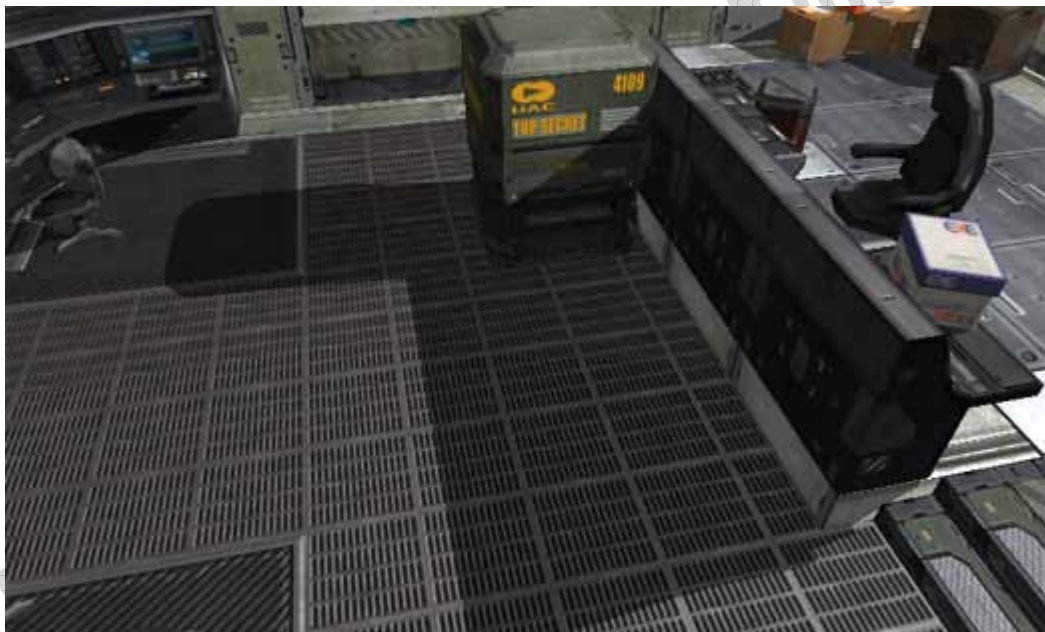
Khi thực hiện Vertex Shader tất cả các vertex không hướng về nguồn sáng sẽ bị đẩy ra xa theo hướng ánh sáng, hình thành vùng bóng tối (shadow volume).

4.2.3. Một số kết quả đạt được

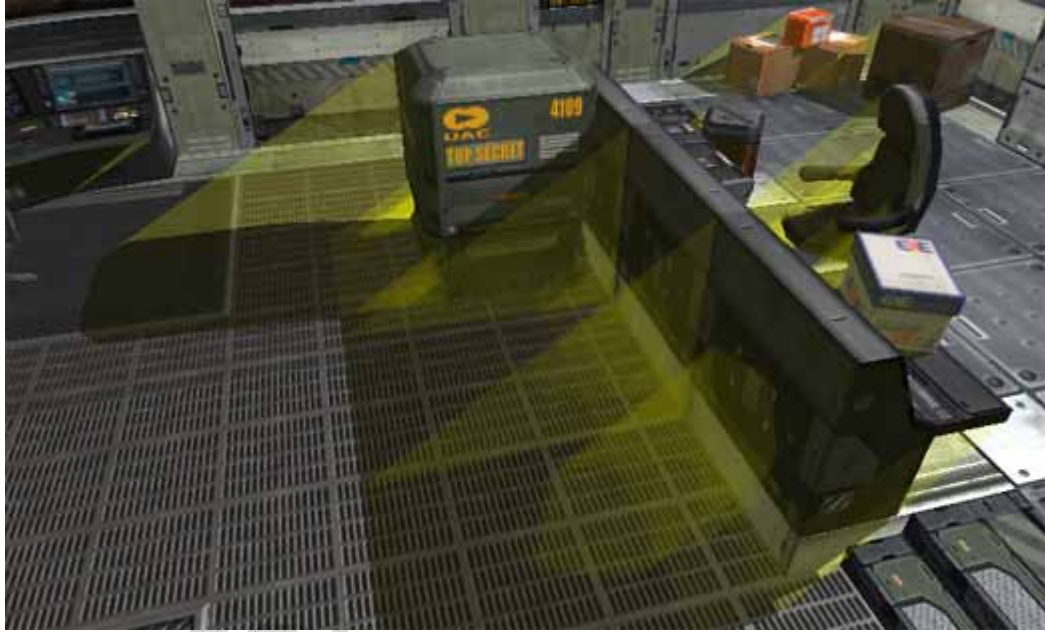
Việc sử dụng hiệu ứng này trong Game demo đạt hiệu quả rất cao do gần với thực tế. Nếu không có đổ bóng, các vật thể có vẻ lơ lửng trong không gian, nhưng nếu có đổ bóng chất lượng hình ảnh đã tăng lên rõ rệt. Các cảnh sau đây được chụp từ Game demo.



Hình 4-6 Bối cảnh không có đổ bóng thời gian thực



Hình 4-7 Bối cảnh có đổ bóng thời gian thực



Hình 4-8 Shadow volume được vẽ bao trùm các vùng tối

4.3. Khung cảnh bầu trời (skybox)

Trong các Game hiện nay các hậu cảnh là điều không thể thiếu vì chúng mang lại chất lượng rất đồ họa rực rỡ, tạo tính thực cho bối cảnh. Cái làm nền cho các hậu cảnh lại chính là khung cảnh bầu trời, nếu không có chúng ta sẽ khó phân biệt được, đâu là ngày, đâu là đêm...

Nhằm góp phần làm cho hậu cảnh gần hơn với thực tế, Nwfc Engine có hỗ trợ thêm Vertex Shader để thực hiện vẽ các khung cảnh bầu trời.

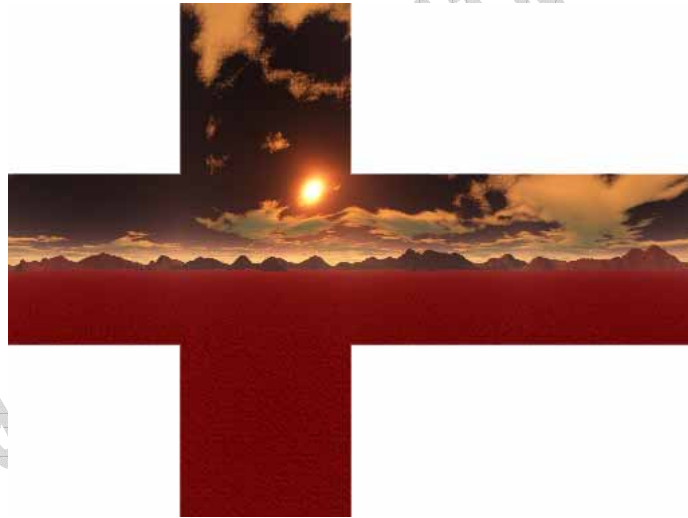
4.3.1. Cơ sở lý thuyết

➤ Cách biểu diễn bầu trời bằng hình khối và texture.

Vì bầu trời là khung cảnh đóng nên các hình khối đóng được sử dụng khá nhiều để thể hiện bầu trời. Các hình khối thường được sử dụng là hình khối vuông (box), hình cầu hay bán cầu (sphere). Các texture được sử dụng để tạo khung cảnh bầu trời thường là các texture biểu hiện 6 mặt của không gian xung quanh (dùng cho box) hay các texture liên nhau ở các cạnh (dùng cho sphere).



Hình 4-9 Texture liên nhau ở các cạnh dùng cho sky sphere



Hình 4-10 Texture 6 mặt dùng cho sky box

➤ Các đặc tính của bầu trời trong thực tế

Khung cảnh bầu trời trong thực tế có các đặc tính đây mà ta cần quan tâm khi muốn thiết kế Vertex Shader.

✚ Rất xa so với tầm nhìn, khi ta nhìn tập trung vào 1 hướng thì dù ta có di chuyển đến đâu đi nữa thì theo hướng nhìn (với điều kiện khoảng cách không quá lớn) thì hình ảnh mà ta nhận được từ bầu trời là không đổi.

✚ Tuy nhiên hình ảnh từ bầu trời mà ta nhận được sẽ thay đổi khi ta nhìn ở các hướng khác nhau.

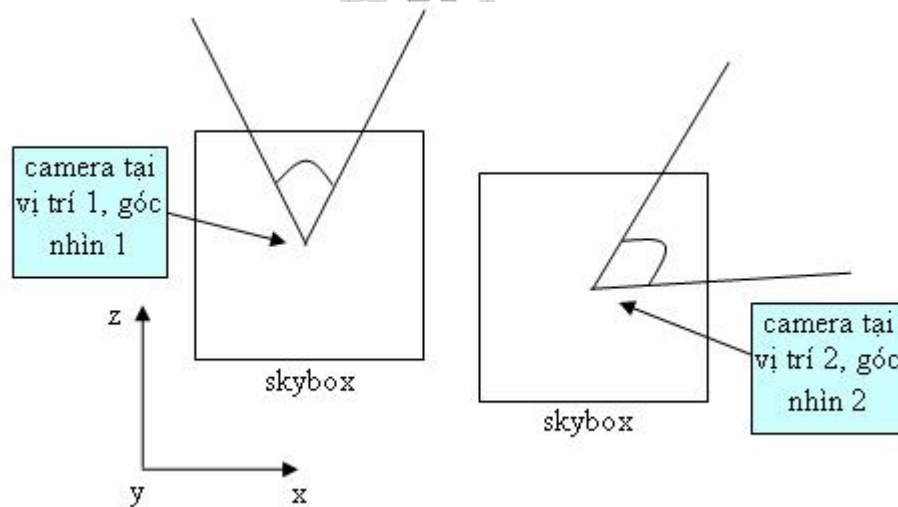
Từ các đặc tính trên của bầu trời ta xác định được cách thức biểu diễn bầu trời trong 3D như sau:

✚ Sử dụng một hình khối để làm vật chứa và sử dụng 1 texture có hình khung cảnh bầu trời.

✚ Vì ta không thể nào đi xuyên qua bầu trời, nên ta phải luôn cập nhật vị trí của hình khối bầu trời = vị trí hiện thời của camera (hay vị trí của mắt) (thỏa mãn tính chất 1).

✚ Chỉ cập nhật vị trí mà không thay đổi góc xoay của hình khối bầu trời nhằm khiến cho hình khối bầu trời không thay đổi theo hướng xoay của camera (thỏa mãn tính chất 2).

Sau đây là hình vẽ minh họa cho ý tưởng.



Hình 4-11 Tọa độ của skybox được cập nhật theo tọa độ camera

4.3.2. Vertex Shader cho skybox

Vertex Shader cho skybox khá đơn giản như sau:

```
struct VS_INPUT {
    float4 position      :    POSITION;
    float2 texcoord      :    TEXCOORD0;
};

struct VS_OUTPUT {
    float4 position      :    POSITION;
    float4 color          :    COLOR0;
    float2 texcoord      :    TEXCOORD0;
};

VS_OUTPUT main( VS_INPUT i )
```

```
{  
    VS_OUTPUT o;  
    // Skybox local transform  
    float3 worldPos = mul( float4( i.position.xyz, 0 ), cModel[0] );  
    // Vertex world position = eye position  
    worldPos += cEyePos;  
    // Transform vertex to projection space  
    float4 projPos = mul( float4( worldPos, 1 ), cViewProj );  
    // Sky is far away, so depth value = 1.0f  
    projPos.z = 0.9999f * projPos.w;  
    o.position = projPos;  
    // Final color  
    o.color = 1.0f;  
    o.texcoord = i.texcoord;  
    return o;  
}
```

Trước tiên ta phải biến đổi sky box bằng ma trận thế giới, việc này có vẻ như là không cần thiết vì ta sẽ sử dụng vị trí của mắt (hay camera) làm vị trí cho skybox. Nhưng thực tế công việc này cho phép ta triển khai 1 số thuộc tính ban đầu cho skybox như độ cao đối với tầm mắt, độ phóng đại...

Sau khi cộng thêm tọa độ của mắt vào, ta phải biến đổi vertex vào không gian chiếu bằng cách nhân với ma trận View * Projection.

Vì skybox ở rất xa nên ta cho độ sâu = 1.0f (để các điểm ảnh của skybox không thể vượt qua giá trị độ sâu của các điểm ảnh khác khi kiểm tra độ sâu (depth test)) công đoạn này phải nhân với projPos.w vì để chuẩn bị cho giai đoạn chuẩn hóa hệ tọa độ thuần nhất sau khi kết thúc Vertex Shader.

4.3.3. Một số kết quả đạt được

Khung cảnh bầu trời được vẽ ra đảm bảo đúng các đặc tính của bầu trời trong thực tế, cho dù ta có di chuyển camera thế nào đi nữa, ta cũng không thể “đi xuyên” qua bầu trời được. Các cảnh sau đây được chụp từ Game demo.



Hình 4-12 Khung cảnh bầu trời chính diện



Hình 4-13 Một góc nhìn khác của bầu trời

4.4. Chiếu sáng theo điểm ảnh (per-pixel lighting) sử dụng normal map và specular map

Hiện nay hiệu ứng chiếu sáng trên từng điểm ảnh được sử dụng khá phổ biến trong các Game nhằm tăng cường chất lượng đồ họa cho Game. Thay vì chiếu sáng theo từng đỉnh vertex, per-pixel lighting cho chất lượng đồ họa cao hơn hẳn do có thể áp dụng nhiều thuật toán mới trong đồ họa 3 chiều như bump bề mặt bằng normal map, phản chiếu bề mặt bằng specular map...

Ứng dụng khi tự thực hiện chiếu sáng trên điểm ảnh bằng Shaders phải giải quyết các tất cả các vấn đề về chiếu sáng như diffuse lighting, specular lighting...

Diffuse lighting trong Engine chủ yếu sử dụng bump bằng normal map và specular lighting chủ yếu sử dụng specular map, 2 thuật toán chiếu sáng này sẽ được trình bày kỹ ở phần này.

4.4.1. Cơ sở lý thuyết

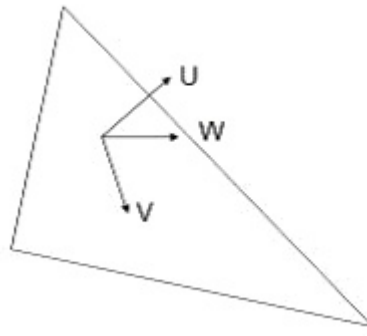
Ở phần này sẽ đề cập chi tiết vào qui trình chiếu sáng trên điểm ảnh được Engine hỗ trợ. Qui trình chiếu sáng trên điểm ảnh chủ yếu phân ra làm 2 công đoạn riêng biệt: thực hiện tính toán màu chính (diffuse color) và tính toán màu phản chiếu (specular color).

➤ Tính toán màu diffuse (có bump bề mặt bằng normal map)

Trước khi đi chi tiết vào thuật toán ta cần xem qua 1 số khái niệm mới dùng trong phần này

✚ Không gian tiếp tuyến của vật thể (tangent space).

Tọa độ texture tại mỗi đỉnh (vertex) hình thành một hệ trục tọa độ 3 chiều với trục U (tiếp tuyến), trục W (pháp tuyến) và trục V (binormal = $U \times W$). Hệ trục tọa độ này gọi là không gian tiếp tuyến hay không gian texture của vật thể tại các đỉnh (vertex).



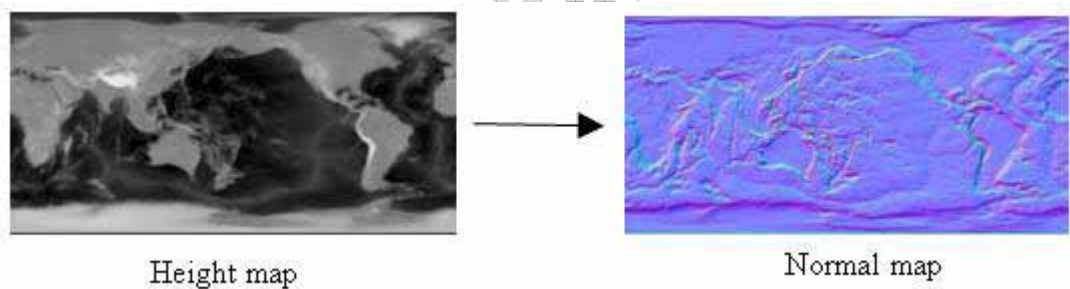
Hình 4-14 Không gian tiếp tuyến

✚ Normal map là gì?

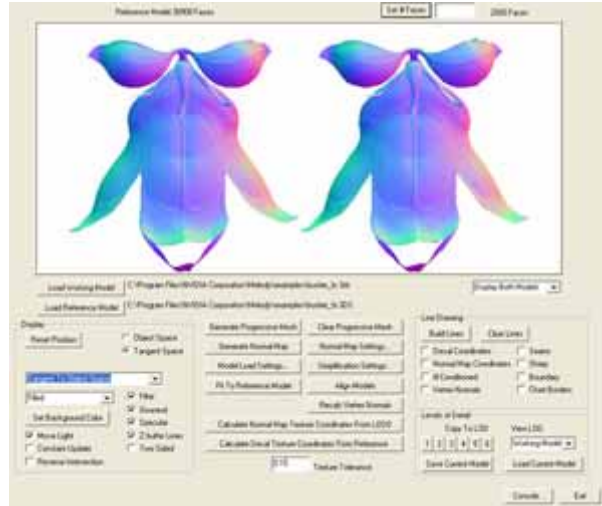
Normal map là một texture nhưng có đặc tính khá đặc biệt, thay vì chứa thông tin về điểm màu như texture thông thường, normal map lại chứa thông tin về không gian tiếp tuyến (tangent space) hay không gian texture (texture space) của vật thể, hay nói cách khác nếu các điểm ảnh của texture biểu diễn màu sắc của vật thể tại 1 điểm thì normal map sẽ biểu diễn không gian tiếp tuyến của vật thể tại điểm đó.

Mỗi điểm ảnh của normal map có định dạng là RGBA trong đó 3 thành phần RGB có giá trị $[0..1]$ được ánh xạ từ 3 trục U, V, W có giá trị trong khoảng $[-1, 1]$.

Normal map có thể tạo ra bằng 2 cách, dùng height map (texture dạng grayscale chứa thông tin độ sâu về bề mặt của vật thể trong đó màu sáng hơn biểu thị độ cao lớn hơn). Cách thứ 2 phức tạp hơn do phải tạo thêm 1 vật thể khác có độ chi tiết cao hơn, sau đó ta so sánh sự khác nhau giữa 2 vật thể để tạo ra normal map (quá trình này có thể được thực hiện bằng tool Melody của NVidia).



Hình 4-15 Tạo normal map từ height map



Hình 4-16 Tạo normal map từ vật thể có độ chi tiết cao hơn bằng Melody (NVIDIA)

✚ Bump bề mặt sử dụng normal map

Bump bề mặt chủ yếu được thực hiện trên Pixel Shader cho từng điểm ảnh. Thuật toán này sử dụng giá trị normal trong normal map để xác định mức độ của ánh sáng tác động vào điểm ảnh đó bằng cách nhân tích vô hướng giá trị normal trên với vector hướng ánh sáng trong không gian tiếp tuyến. Sau đó giá trị này được nhân với màu sắc của vertex và màu lấy mẫu từ texture để tính ra màu diffuse (màu của vertex được tính trong Vertex Shader)

light factor = dot product (normal, light vector)
diffuse color = light factor * vertex color * texture color;

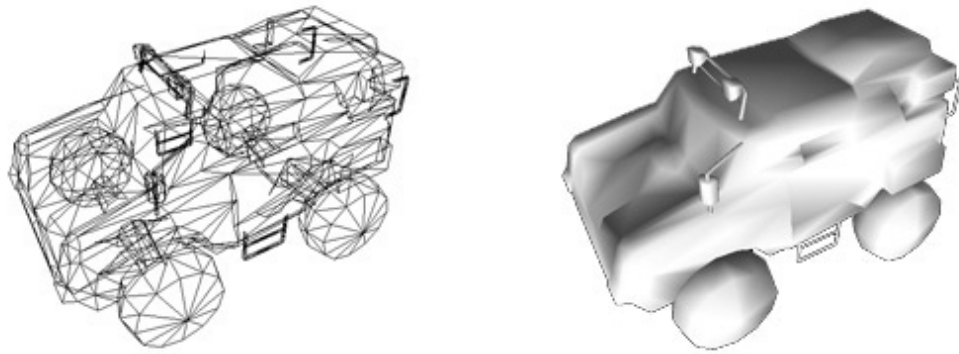
Trong đó

✚ **normal**. Vector pháp tuyến (normal vector) tại điểm đó, normal vector có được do lấy mẫu từ normal map.

✚ **light vector**. Vector hướng ánh sáng trong không gian tiếp tuyến (tangent space), vector này được tính trong Vertex Shader và được truyền vào Pixel Shader để sử dụng.

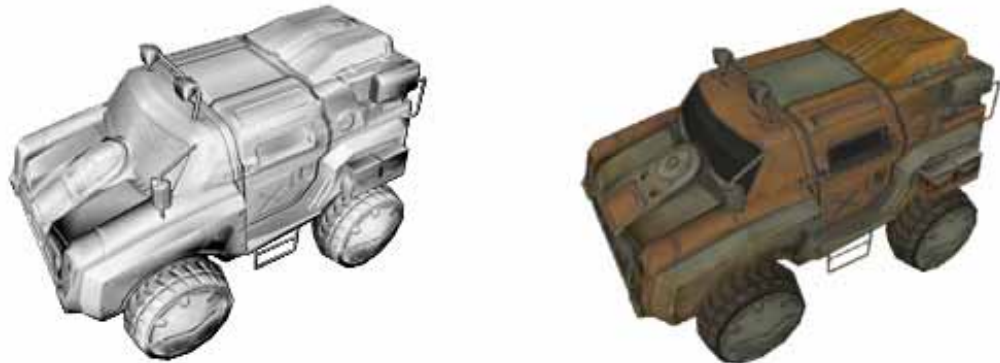
✚ **vertex color**. Màu của vertex sau khi thực hiện chiếu sáng trên vertex (per-vertex lighting) trong Vertex Shader.

✚ **texture color**. Màu của texture chính, có được do lấy mẫu texture.



Hình 4-17 Chiếu sáng theo từng vertex trong Vertex Shader

- ✚ **Hình 1.** Dữ liệu vertex trong bộ nhớ (được vẽ dưới dạng wireframe)
- ✚ **Hình 2.** Chiếu sáng trên từng đỉnh (per-vertex lighting) bằng Vertex Shader



Hình 4-18 Chiếu sáng trên từng điểm ảnh trong Pixel Shader

- ✚ **Hình 1.** Chiếu sáng trên từng pixel (sử dụng tích vô hướng giữa normal và vector hướng ánh sáng).
- ✚ **Hình 2.** Sau khi kết hợp với lấy mẫu từ texture chính.

➤ **Tính toán màu specular (sử dụng specular map)**

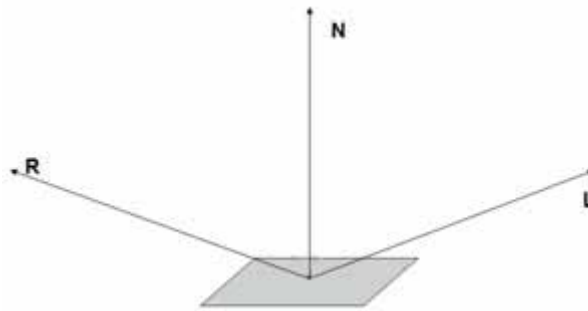
- ✚ **Specular map là gì ?**

Specular map là texture dạng grayscale, specular map có tác dụng cho biết vùng nào của vật thể phản chiếu nhiều ánh sáng vùng nào phản chiếu ít ánh sáng (tương ứng với màu trong specular map từ sáng tới tối).

- ✚ **Tính độ phản chiếu của ánh sáng**

Độ phản chiếu (phản xạ) của ánh sáng trên vật thể thì phụ thuộc vị trí của mắt (hay camera). Khi mắt nằm ngay trên đường phản xạ của ánh sáng thì mắt sẽ nhìn

thấy một vùng ánh sáng chói do toàn bộ năng lượng của ánh sáng được truyền thẳng vào mắt.



Hình 4-19 Sự phản xạ của tia sáng trên bề mặt

Muốn tính màu specular của điểm ảnh ta phải xác định được mức độ ánh sáng phản chiếu tại điểm đó. Công thức tính vector phản chiếu (phản xạ) như sau:

$$\mathbf{R} = 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N} - \mathbf{L}$$

Trong đó:

✚ **L.** Light vector

✚ **R.** Reflection vector

✚ **N.** Normal

Mức độ phản chiếu của ánh sáng phụ thuộc rất nhiều vào chất liệu bề mặt của vật thể, các bề mặt nhẵn bóng có độ phản chiếu lớn trong khi các bề mặt gồ ghề lại có độ phản chiếu thấp. Để tránh công việc phải phân rã vật thể ra thành nhiều thành phần để dựng hình với các mức phản chiếu khác nhau người ta dùng specular map như một lookup table để xác định mức độ phản chiếu của ánh sáng trên từng điểm ảnh.

```
reflection vector = 2 * dotproduct (normal, light vector) * normal - light vector  
specular factor = dotproduct (reflection vector, view vector)  
specular color = (specular factor ^ specular constant) * specular lookup
```

Trong đó

✚ **normal.** Vector pháp tuyến (normal vector) tại điểm đó, normal vector có được do lấy mẫu từ normal map.

✚ **light vector.** Vector hướng ánh sáng trong không gian tiếp tuyến (tangent space), vector này được tính trong Vertex Shader và được truyền vào Pixel Shader để sử dụng.

✚ **view vector.** Vector tính từ mắt đến điểm nhìn (tọa độ trong không gian tiếp tuyến), vector này được tính trong Vertex Shader và truyền vào Pixel Shader để sử dụng.

✚ **specular constant.** Hằng phản chiếu, giá trị càng lớn thì vùng phản chiếu càng nhỏ.

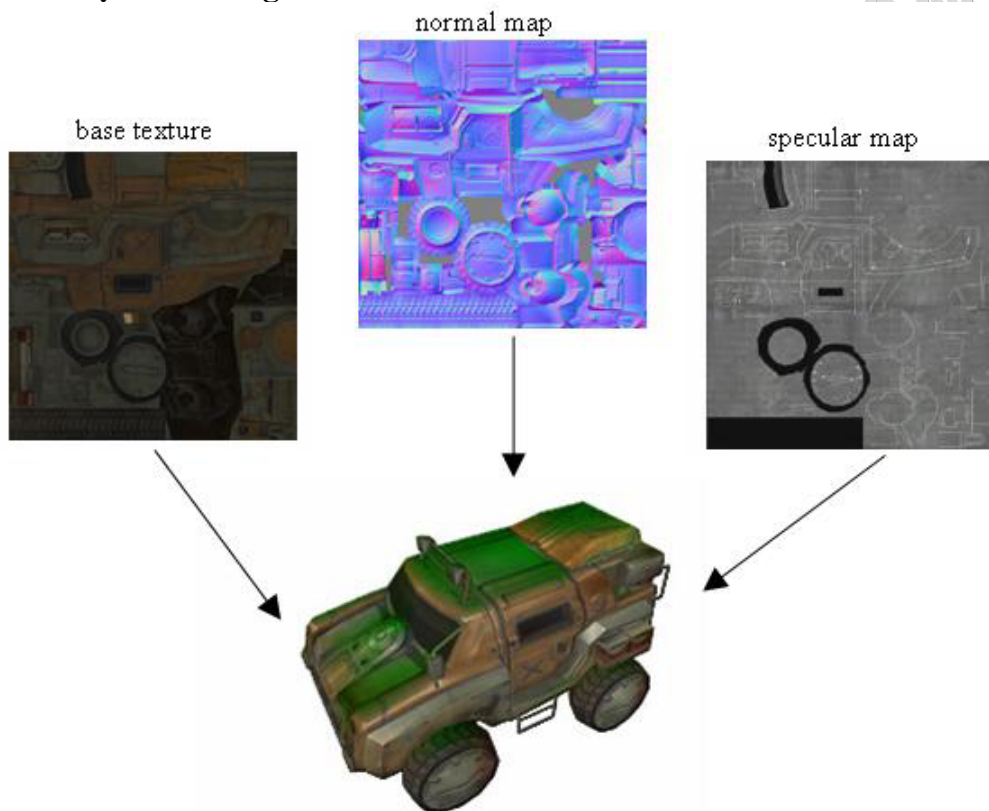
✚ **specular lookup.** Cho biết mức độ phản chiếu của điểm đó, giá trị này có được do lấy mẫu từ specular map.

Sau khi thêm độ phản chiếu ánh sáng vào xe trông như là được cấu tạo từ kim loại, do đó sự phản chiếu góp phần tăng đáng kể chất lượng đồ họa.



Hình 4-20 Tính độ phản chiếu trên từng điểm ảnh

Tóm tắt thuật toán bằng hình vẽ



Hình 4-21 Tóm tắt qui trình per-pixel lighting bằng hình vẽ

4.4.2. Vertex Shader và Pixel Shader cho per-pixel lighting

Thuật toán chiếu sáng trên điểm ảnh (per-pixel lighting) cần cả Vertex Shader và Pixel Shader đi liền với nhau để thực hiện.

➤ Vertex Shader

Vertex Shader dùng ở đây là Vertex Shader phụ thuộc nguồn sáng nên cần được biên dịch thành 11 Vertex Shader tương ứng với 11 light combo.

```
...
struct VS_INPUT {
    float4 position      :    POSITION;
    float3 normal        :    NORMAL;
    float2 texcoord0     :    TEXCOORD0;
    float3 tangent       :    TANGENT;
    float3 binormal      :    BINORMAL;
};
struct VS_OUTPUT {
    float4 position      :    POSITION;
    float4 color         :    COLOR0;
    float4 ambient       :    COLOR1;
    float2 baseTexCoord  :    TEXCOORD0;
    float3 light_vector  :    TEXCOORD1;
    float3 view_vector   :    TEXCOORD2;
};
VS_OUTPUT main( const VS_INPUT i )
{
    VS_OUTPUT o = ( VS_OUTPUT )0;
    // Calculate world vertex's elements and lighting
    float3 worldPos = . . .
    float3 worldNormal = . . .
    o.ambient = . . .
    o.color = . . .

    // Calculate project space position
    o.position = mul( float4( worldPos, 1 ), cViewProj );

    // Calculate the light vector in object space,
    // and then transform it into texture space.
    float3 temp_light_vector;
    if( g_LocalLightType0 == LIGHTTYPE_DIRECTIONAL )
        temp_light_vector = mul( -cLightInfo[0].dir, cInvModel );
    else
        temp_light_vector = mul( cLightInfo[0].pos, cInvModel )-i.position;
```

```
o.light_vector.x = dot( temp_light_vector, i.tangent );
o.light_vector.y = dot( temp_light_vector, i.binormal );
o.light_vector.z = dot( temp_light_vector, i.normal );

// Calculate the view vector in object space,
// and then transform it into texture space.
float3 temp_view_vector = mul(cEyePos, cInvModel) - i.position;
o.view_vector.x = dot( temp_view_vector, i.tangent );
o.view_vector.y = dot( temp_view_vector, i.binormal );
o.view_vector.z = dot( temp_view_vector, i.normal );
// Pass texture coord 0 to output
o.baseTexCoord = i.texcoord0;
return o;
}
```

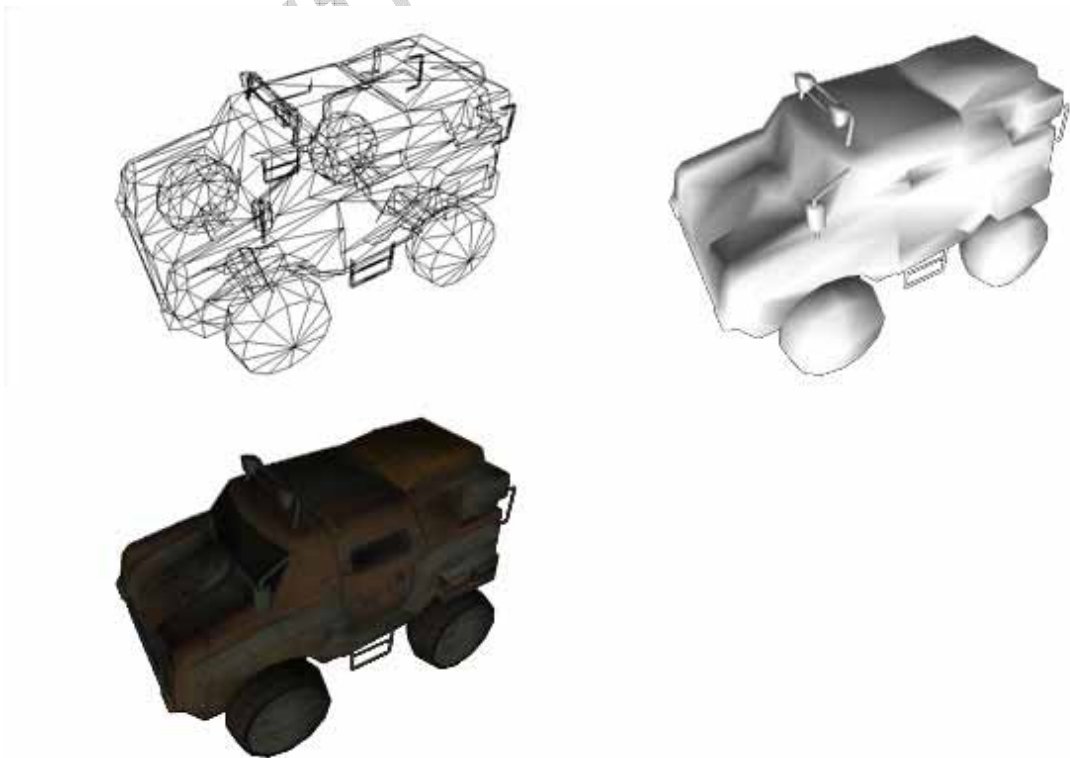
Pixel Shader

```
...
struct PS_INPUT {
    float4 color      :    COLOR0;
    float4 ambient    :    COLOR1;
    float2 baseTexCoord :    TEXCOORD0;
    float3 light_vector :    TEXCOORD1;
    float3 view_vector  :    TEXCOORD2;
};
struct PS_OUTPUT {
    float4 color :    COLOR0;
};
PS_OUTPUT main( PS_INPUT i )
{
    PS_OUTPUT o;
    // Sampler normal value from normal map
    float3 bump = tex2D( bumpSampler, i.baseTexCoord );
    float3 localNormal = normalize( (bump * 2.0f) - 1.0f );
    // Normalize light vector and view vector
    float3 normalize_light_vector = normalize( i.light_vector );
    float3 normalize_view_vector = normalize( i.view_vector );
    // Calculate light factor by dot normal with light vector
    float4 n_dot_l = dot( localNormal, normalize_light_vector );
    // Calculate the specular reflection vector from light and normal
    float3 haft_angle=dot(normalize_light_vector,localNormal)*localNormal;
    float3 reflection_vector = haft_angle* 2.0f - normalize_light_vector;
    // Calculate the specular factor and specular color
    float specularFactor =
        pow( dot(normalize_view_vector, reflection_vector), 2.0f );
    // Sampler specular lookup table value from specular map
    float3 specular = tex2D(specularSampler,i.baseTexCoord) * 2.0f;
```

```
// Sampler base texture
float4 base = tex2D( baseSampler, i.baseTexCoord );
base.rgb *= 1.2;
// Calculate final color
o.color.rgb=(( specularFactor * specular) + base) *
             saturate(n_dot_l) * i.color;
o.color.rgb += saturate( dot(localNormal, bump_ambient) ) *
               base * i.ambient * saturate(1.0 - n_dot_l);
// Set the alpha component by base map for alpha supporting
o.color.a = base.a;
return o; // Return the resulting output struct
}
```

4.4.3. Một số kết quả đạt được

Các ảnh sau đây được chụp từ Engine ở thời gian thực.

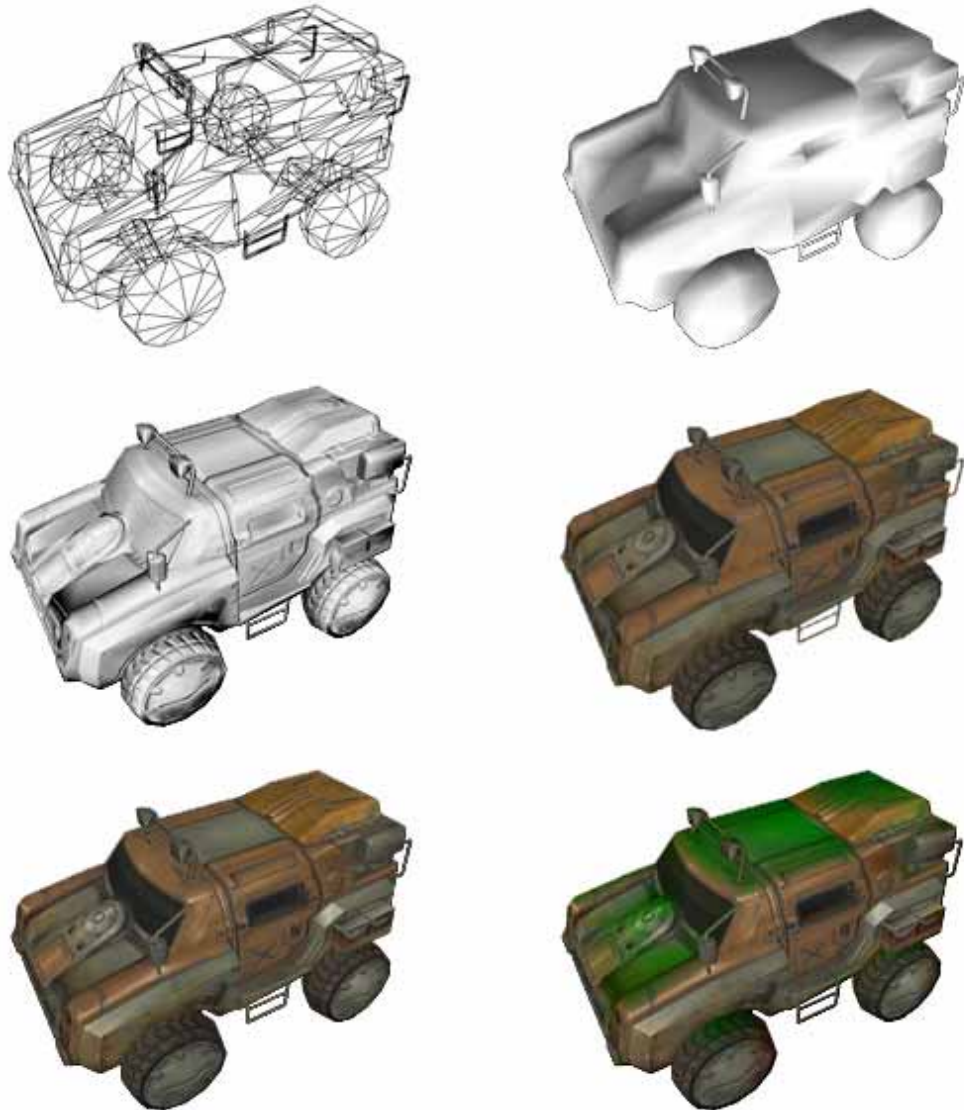


Hình 4-22 Các công đoạn sử dụng Fixed Function

Xe khi được dựng hình bằng Fixed Function, do texture không được chiếu sáng trước nên xe dựng được rất tối, hầu như ta không thể thấy được chi tiết của xe. Giải thích các giai đoạn dựng hình (từ trái qua phải, từ trên xuống dưới).

- ✚ Các đỉnh (vertices) trong bộ nhớ có dạng wire frame như trên hình.
- ✚ Xe sau khi đã được chiếu sáng trên từng đỉnh (per-vertex lighting).

- ✚ Xe sau khi xử lý pixel và áp texture (bằng Fixed Function Pipeline).

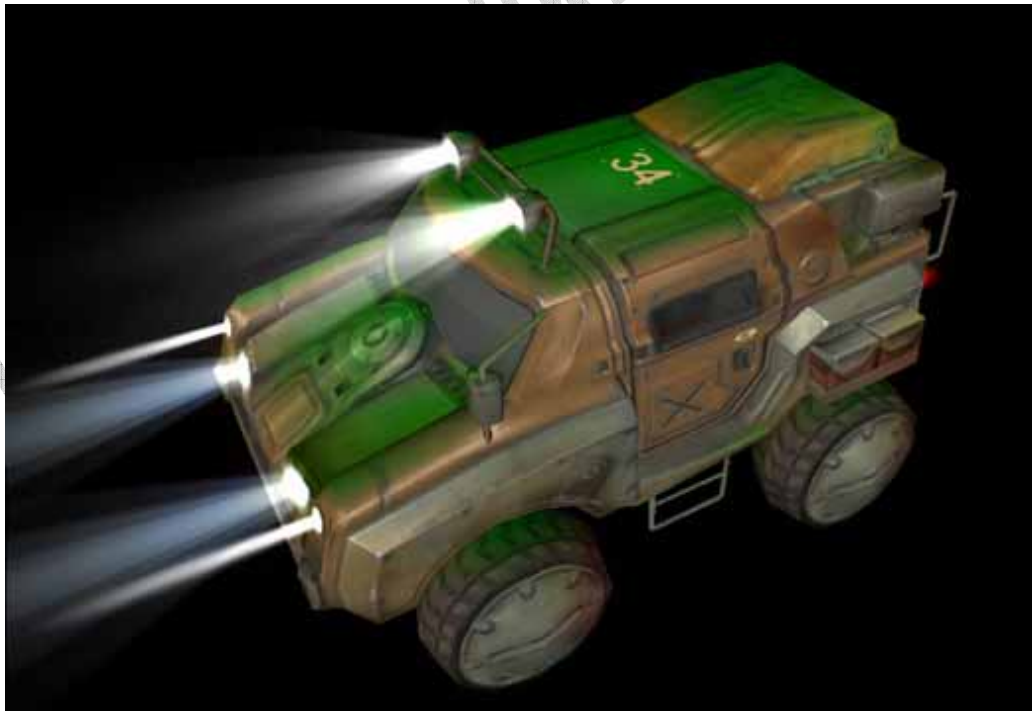


Hình 4-23 Các công đoạn sử dụng Shaders per-pixel lighting

Xe khi được dựng hình bằng Vertex Shader và Pixel Shader bằng cách chiếu sáng trên điểm ảnh (per-pixel lighting) chất lượng hình ảnh cao hơn hẳn. Texture được chiếu sáng trên từng điểm ảnh nên ta có thể thấy được độ hầu hết các chi tiết của xe. Giải thích các giai đoạn dựng hình (từ trái qua phải, từ trên xuống dưới).

- ✚ Các đỉnh (vertices) trong bộ nhớ có dạng wire frame như trên hình.
- ✚ Xe sau khi đã được chiếu sáng trên từng đỉnh (per-vertex lighting) theo nguồn sáng.

- ✚ Xe sau khi xử lý pixel, thực hiện chiếu sáng trên từng điểm ảnh dùng normal map.
- ✚ Xe sau khi áp texture bề mặt. Ta thấy xe vẫn còn thiếu một số độ phản chiếu ánh sáng.
- ✚ Xe sau khi thực hiện ánh sáng phản chiếu dùng specular map.
- ✚ Xe sau khi thực hiện kết hợp với ánh sáng môi trường 6 mặt (ambient light cube).



Hình 4-24 Kết quả sau cùng sau khi bổ sung một số hiệu ứng

4.5. Tóm tắt

Trong chương 4 chúng tôi đã trình bày chi tiết các thuật toán Vertex Shader và Pixel Shader dùng để dựng hình 3D trong Nwfc Engine. Các thuật toán này đảm bảo chất lượng dựng hình của Nwfc Engine như thật và thời gian thực hiện nhanh.

Hai chương tiếp theo (5 và 6) sẽ trình bày về việc tìm hiểu và xây dựng 2 hệ thống diễn hoạt và vật lý. Đây là 2 hệ thống xây dựng để kết hợp với Nwfc Engine đủ để tạo ra ứng dụng Game hoàn chỉnh.

Chương 5 Hệ thống diễn hoạt (Animation System)

- ◇ [Giới thiệu hệ thống diễn hoạt](#)
- ◇ [Các vấn đề cần giải quyết](#)
- ◇ [Hiện thực hệ thống diễn hoạt](#)
- ◇ [Tóm tắt](#)

5.1. Giới thiệu hệ thống diễn hoạt

Hệ thống diễn hoạt là hệ thống giúp quản lý và thực hiện tất cả các diễn hoạt cho các đối tượng trong Game. Một hệ thống diễn hoạt tốt phải đảm bảo cho phép quản lý chính xác, dễ dàng đến các diễn hoạt của từng đối tượng, và đảm bảo xử lý được các tình huống diễn hoạt có trong Game như kết hợp diễn hoạt, nhảy, xoay... Hệ thống phải cho phép các hệ thống khác truy cập dữ liệu dễ dàng vì hệ thống này liên quan chặt chẽ đến các hệ thống khác. Hệ thống diễn hoạt còn phải có chất lượng cao, xử lý nhanh và không chiếm nhiều bộ nhớ sử dụng.

Dựa trên những đặc điểm và yêu cầu của một hệ thống diễn hoạt chúng tôi đã xây dựng một hệ thống diễn hoạt cung cấp các tính năng sau:

- ✚ Đọc và xử lý dữ liệu khung xương và diễn hoạt cho các nhân vật từ một tập tin dữ liệu có định dạng md5.
- ✚ Quản lý cấu trúc khung xương hiệu quả cho các nhân vật đảm bảo được tính mở rộng. Cấu trúc khung xương được tổ chức theo một cây với mỗi nút trong cây lưu các thông tin biến đổi cho các khớp xương của khung xương.
- ✚ Quản lý thông tin diễn hoạt cho từng đối tượng.
- ✚ Thực hiện kết hợp nhiều diễn hoạt lại vào trên cùng một đối tượng và kết hợp các diễn hoạt trong các phần khác nhau của đối tượng.
- ✚ Thực hiện xây dựng nên các đường điều khiển nhằm có thể điều khiển nhân vật đi theo những lộ trình định trước.

5.2. Các vấn đề cần giải quyết

Trước khi đi vào cài đặt hệ thống chúng ta cùng xem xét các vấn đề cần giải quyết và cách giải quyết chúng. Việc cài đặt các lớp cụ thể sẽ dựa trên việc phân tích và các hướng giải quyết này.

5.2.1. Tập tin lưu dữ liệu diễn hoạt

Phần đầu tiên và quan trọng của một hệ thống diễn hoạt là xây dựng hoặc chọn một định dạng tập tin lưu trữ dữ liệu diễn hoạt cho các đối tượng. Sau khi tìm hiểu và chọn lọc, chúng tôi chọn định dạng **md5** để lưu trữ dữ liệu diễn hoạt cho hệ thống của mình.

5.2.1.1. Tập tin md5

➤ **Tổng quan về tập tin định dạng md5:**

Md5 là một định dạng tập tin được dùng để lưu thông tin diễn hoạt cho các nhân vật trong Game DOOM3. Đây là một định dạng mới được xây dựng và đang có phiên bản thứ 10. Các thông tin lưu trong tập tin md5 ở dạng văn bản và được tổ chức rõ ràng, dễ hiểu với người sử dụng. Với định dạng tập tin này, thông thường các tập tin liên quan đến một nhân vật được để trong cùng thư mục. Trong thư mục này có một tập tin **<tên nhân vật>.md5mesh** lưu thông tin về cấu trúc khung xương và các mesh còn mỗi tập tin **<tên diễn hoạt>.md5anim** lưu thông tin về một diễn hoạt. Ví dụ ta có nhân vật monster thì ta có thư mục monster có một số tập tin như **monster.md5mesh**, **idle.md5anim**, **walk.md5anim**, ...

Cụ thể thông tin trong được lưu trong các tập tin md5 như sau:

➤ **Tập tin .md5mesh:**

Tập tin này lưu thông tin về các khớp (joint) và các mesh. Tập tin sẽ chứa danh sách các khớp hình thành nên cấu trúc khung xương. Mỗi khớp có các thông tin về nhãn, khớp cha của nó, vị trí, hướng quay.

Ví dụ: **"origin" -1 (0 0 0) (-0.7071067095 0 0)** đây là khớp có

- ✚ Nhãn là "origin".
- ✚ Khớp cha mà nó liên kết đến là -1 (có nghĩa nó không có cha).
- ✚ Vị trí X, Y, Z lần lượt là 0, 0, 0.

✚ Hướng sử dụng 3 giá trị thực là các thành phần x, y, z của một quaternion. Một quaternion có 4 thành phần x, y, z, w, do quaternion có độ dài đơn vị nên ta có thể tính w như sau:

```
float tmp = 1.0f - ( (x*x) + (y*y) + (z*z) );  
if(tmp >= 0)  
    w = - (float) sqrt( tmp );  
else  
    w = 0;
```

Ngoài thông tin về khung xương thì tập tin còn lưu thông tin về các mesh. Mỗi mesh bao gồm các đỉnh, các tam giác, và các trọng số (weight).

✚ Ví dụ: **vert 0 (0.0306090005 0.6130819917) 0 2**) chứa các thông tin về mesh như sau: thông tin về tọa độ texture gồm 2 số thực $u = 0.0306090005$ và $v = 0.6130819917$ và giới hạn của trọng số (sử dụng chỉ mục vào giá trị trong danh sách các trọng số và một số đếm).

✚ Các tam giác (ví dụ: **tri 0 2 1 0**) được hình thành bằng cách chỉ ra 3 đỉnh trong danh sách đỉnh.

✚ Các trọng số (ví dụ: **weight 0 5 0.7900747061 (1.9485528469 7.4760251045 23.7117900848)**) chỉ ra khớp liên quan cùng với trọng số của nó. Chú ý rằng tổng số của tất cả các trọng số cho một đỉnh bằng 1. Để tìm ra vị trí của một đỉnh trong 3D ta tính trọng số trung bình của vị trí liên quan của các đỉnh bao lấy nó.

➤ Tập tin **.md5anim**:

Mỗi tập tin **.md5anim** xác định một chuyển động cho một đối tượng như đi, chạy, nhảy, chết, ... Về cơ bản, ta có thể sử dụng lại diễn hoạt cho các mô hình khác với khung xương đủ tương xứng.

Tập tin **.md5anim** xác định ra một cấu trúc cây kế thừa của các khớp, hộp bao quanh nhân vật chỉ ra ranh giới để kiểm tra va chạm, một khung hình cơ sở (base frame) và các khung hình diễn hoạt chính (animation frame). Cấu trúc kế thừa của các khớp giống với cấu trúc khung xương được xác định trong tập tin **.md5mesh**

của đối tượng. Mỗi khớp được xác định bằng một cờ nhị phân cho biết biến số của khớp có thay đổi trong diễn hoạt hay không. Có 6 biến số liên quan đến sự chuyển dịch (translation) X/Y/Z, và sự định hướng (orientation) X/Y/Z. Như vậy cờ sẽ có giá trị sẽ nằm trong khoảng từ 0 đến 63 ($= 2^6 - 1$).

Trong tập tin còn có các giá trị về các hộp bao xung quanh giúp ta có thể kiểm tra nhanh sự va chạm giữa đối tượng này với các vật thể và đối tượng khác. Nhờ việc sử dụng các hộp bao bọc mà ta có thể nhận biết được ngay các mô hình không xảy ra va chạm. Và tất nhiên các giá trị này không liên quan đến việc hiển thị.

Khung hình cơ bản (base frame) được dùng làm khung bắt đầu cho diễn hoạt. Mỗi khung hình diễn hoạt được xác định bằng việc thay thế giá trị của các biến của các khớp liên quan dựa theo khung hình cơ sở. Nếu giá trị của các khớp không thay đổi thì sẽ lấy giá trị từ khung hình cơ sở. Chính vì vậy mà khung cơ sở chính là một điểm mốc cho quan trọng trong diễn hoạt.

Thông tin về diễn hoạt được lưu trữ trong danh sách các giá trị thực trong mỗi khung chuyển động. Đó là các giá trị phải được thay thế dựa vào khung cơ sở cho phù hợp với giá trị các cờ.

5.2.1.2. Xử lý dữ liệu tập tin md5

➤ Mục đích

Sau khi đã tìm hiểu cấu trúc của tập tin md5 ta cần phải đọc và xử lý được dữ liệu trong tập tin để có thể sử dụng trong chương trình.

➤ Cách giải quyết

Trước hết ta phải xây dựng nên các cấu trúc dữ liệu lưu trữ dữ liệu đủ tổng quát để khi cần ta có thể thay đổi được định dạng tập tin lưu trữ mà không ảnh hưởng đến logic chương trình. Các dữ liệu 3D đặc trưng cần phải nạp từ tập tin md5 phải tổ chức thành các dữ liệu cấu trúc:

✚ Các dữ liệu về khớp: chỉ số, tên, chỉ số khớp cha, tên khớp cha, vector chỉ vị trí ban đầu, quaternion xác định góc quay ban đầu.

✚ Dữ liệu về các đỉnh: u, v (tọa độ texture), chỉ số trọng số và số trong số kết nối với đỉnh.

✚ Dữ liệu về các mặt: gồm chỉ số của 3 đỉnh hình thành nên một mặt.

✚ Dữ liệu về trọng số: chỉ số của khớp và trọng số tương ứng.

✚ Dữ liệu mesh: tên, danh sách đỉnh, danh sách mặt, danh sách các trọng số.

✚ Dữ liệu cho các khung hình: bao gồm vector và quaternion cho mỗi khung hình.

✚ Dữ liệu diễn hoạt: khớp hiện tại, khớp cha, danh sách khung hình diễn hoạt, khung hình cơ sở, thuộc tính md5 của khớp (nhằm xác định sự thay đổi dữ liệu các khung hình so với khung hình cơ sở).

Việc lưu lại thông tin nạp từ tập tin một cách tinh tế, hiệu quả còn giúp cho ta tiết kiệm bộ nhớ rất lớn. Giả sử như ta có 2 con quái vật giống hệt nhau cùng xuất hiện trong màn Game thì 2 con quái vật này phải cùng truy xuất đến một dữ liệu đã nạp. Có nghĩa là nếu dữ liệu cho con quái vật đã nạp từ trước thì ta không cần nạp nữa mà chỉ truy xuất đến dữ liệu dùng chung đã nạp mà thôi. Nếu ta không chú ý đến vấn đề này thì sẽ gây lãng phí rất lớn về bộ nhớ và thời gian nạp dữ liệu. Để giải quyết thì ta chỉ thực hiện nạp dữ liệu một lần (vào lần đầu tiên yêu cầu dữ liệu), khi cần dùng ta dùng các con trỏ chỉ đến dữ liệu và xử lý. Ta không được thay đổi các dữ liệu gốc nạp từ tập tin để đảm bảo không ảnh hưởng đến các đối tượng đang sử dụng chung dữ liệu.

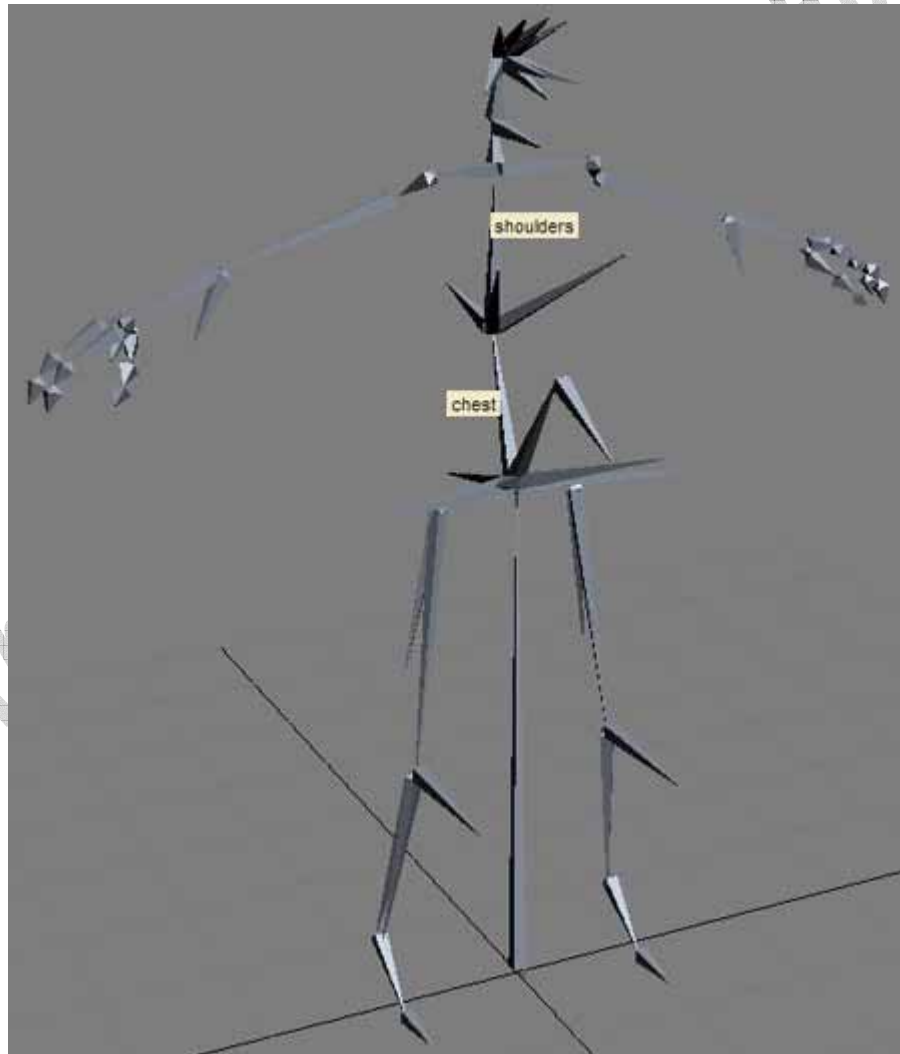
Việc đọc dữ liệu sẽ được đọc theo trình tự bố trí dữ liệu trong tập tin md5 với sự hỗ trợ của một công cụ cho phép ta phân tích tập tin để tìm thông tin chính xác và nhanh.

5.2.2. Vấn đề về khung xương

5.2.2.1. Giới thiệu về khung xương

Các đối tượng mà hệ thống diễn hoạt điều khiển thông thường có cấu trúc là một khung xương. Khung xương là một cấu trúc cây trong đó mỗi nút tương ứng với

một khớp xương. Mỗi nút trong khung xương sẽ có một nút cha (trừ nút đầu tiên) và có thể có nhiều nút con. Hình dưới là một ví dụ về khung xương:



Hình 5-1 Ví dụ cấu trúc khung xương

Khi đối tượng chuyển động thì các xương của nó chuyển động. Trong cấu trúc khung xương thì chuyển động chủ yếu là chuyển động quay ví dụ như ta đưa tay lên thật chất là ta quay xương cánh tay quanh khuỷ tay một góc. Trong cấu trúc khung xương có một tính chất biến đổi quan trọng là sự chuyển động của một khớp sẽ được kế thừa chuyển động từ khớp cha của nó. Như vậy khi cần cập nhật lại khung xương thì ta phải cập nhật từ nút gốc trở đi. Nếu ta thực hiện biến đổi trên nút gốc thì biến đổi đó ảnh hưởng toàn khung xương, ví dụ như ta dịch chuyển nút gốc đi vị trí khác thì toàn bộ khung xương sẽ bị dịch theo.

5.2.2.2. Tổ chức dữ liệu

Vấn đề đầu tiên là ta phải tổ chức dữ liệu cho khung xương làm sao để ta có thể tách rời các phần của cơ thể ví dụ như phần thân, phần đầu ra thành các phần độc lập nhau? Việc phân tách giúp ta có thể điều khiển từng thành phần riêng biệt và đặc biệt ta có thể thay đổi hình dạng cho nhân vật. Chúng ta có thể gắn các đầu khác nhau vào trong cùng một thân hình để tạo ra các nhân vật khác nhau. Và làm sao ta có thể gắn thêm các phần khác vào trong cấu trúc khung xương? Ví dụ như khi nhân vật cầm một khẩu súng thì làm sao ta có thể gắn khẩu súng này vào tay nhân vật và chuyển động của khẩu súng khớp với chuyển động của tay.

Để giải quyết được vấn đề này ta sẽ cho các nút trong cấu trúc cây của khung xương kế thừa từ một cấu trúc dữ liệu đặc biệt chỉ chứa các dữ liệu biến đổi. Việc tổ chức dữ liệu này tương tự như khái niệm Scene Graph được dùng rất nhiều trong việc quản lý Game. Một nút biến đổi (transform node) sẽ lưu các thông tin sau:

- ✚ Tên.
- ✚ Nút cha của nó.
- ✚ Vị trí và góc quay ban đầu.
- ✚ Vị trí và góc quay trong môi quan hệ với nút cha.
- ✚ Vị trí, góc quay và ma trận biến đổi so với thể giới (vị trí và góc quay thật sự trong cảnh 3D).

Việc biến đổi của các khớp xương chính là sự biến đổi của các nút biến đổi. Bây giờ ta có thể tạo ra các cấu trúc khung xương riêng cho phần thân, phần đầu. Sau đó ta sẽ gắn đầu vào thân bằng cách ta thực hiện việc gán nút đầu tiên của đầu là nút con của đốt xương cổ của phần thân.

Tương tự như vậy, nếu ta muốn cho nhân vật cầm một cây súng thì ta cũng tạo ra một cấu trúc khung xương cho cây súng. Sau đó ta thực hiện việc thiết lập nút cha của cây súng là một khớp xương ở tay, bây giờ biến đổi của cây súng sẽ phụ thuộc vào biến đổi của khớp xương tay.

Rõ ràng việc tổ chức dữ liệu như vậy giúp ta giải quyết được nhiều vấn đề và giúp ta quản lý các đối tượng dễ dàng và thống nhất. Một cách tổng quát, ta có thể tạo một cây kế thừa kết nối tất cả các đối tượng có cấu trúc khung xương. Việc tìm kiếm, cập nhật và hiển thị sẽ đều thực hiện trên cây chung này. Đây cũng chính là ý tưởng và cách thức thực hiện của một Scene Graph.

Vấn đề tiếp theo là làm sao ta có thể thực hiện việc cập nhật và truy xuất các khớp trong khung xương nhanh chóng và phù hợp với việc tổ chức dữ liệu trên tập tin md5?

Dựa vào đặc điểm tập tin md5 lưu các khớp trong một khung xương theo một trình tự tăng dần của các khớp xương ta có thể có cách tổ chức dữ liệu tương ứng. Nếu ta tổ chức các khớp trong khung xương là các cấu trúc trong đó có các con trỏ chỉ đến các nút con thì đây là cách tổ chức sát với khái niệm của cấu trúc xương. Tuy nhiên việc tổ chức như vậy thì cứ mỗi lần cập nhật hoặc tìm kiếm một khớp xương theo tên hoặc theo chỉ số định danh ta cần phải duyệt cây theo đệ quy. Do thao tác tìm kiếm và duyệt như vậy dùng với tần suất rất lớn sẽ làm cho chi phí tăng nhanh. Để khắc phục vấn đề này, ta đơn giản sẽ tổ chức các khớp của khung xương thành một mảng. Khi đó việc tìm kiếm một khớp xương theo chỉ số ta sẽ thực hiện truy xuất thẳng đến phần tử có chỉ số tương ứng trong mảng. Như vậy, với một cải tiến nhỏ ta đã giảm đi được rất nhiều chi phí trong việc tìm kiếm.

5.2.2.3. Cập nhật và di chuyển khung xương

Mỗi một nhân vật là một cấu trúc khung xương. Trong cấu trúc khung xương này ta có một nút là nút gốc (nút này không có nút cha). Mọi thao tác xử lý trên khung xương phải được bắt đầu từ nút gốc này.

Việc cập nhật lại khung xương phải được thực hiện thường xuyên để phù hợp với diễn hoạt. Việc cập nhật đơn giản ta sẽ thực hiện việc cập nhật lại vị trí và góc quay hay là ma trận biến đổi cho từng khớp trong khung xương bắt đầu từ nút gốc. Biến đổi của một nút sẽ bằng biến đổi của nút cha nhân với biến đổi của nó với nút cha.

Bên cạnh việc biến đổi trong cấu trúc khung xương thì ta cần toàn bộ khung xương biến đổi như di chuyển vị trí, thay đổi góc quay so với lại thế giới trong Game. Để thực hiện việc đó ta phải có các biến để lưu vị trí và góc quay hiện thời của nút gốc. Ta sẽ thực hiện việc cập nhật toàn bộ khung xương với từ nút gốc với vị trí và góc quay là vị trí và góc quay hiện thời đang lưu.

5.2.3. Đường dẫn định hướng cho diễn hoạt

5.2.3.1. Giới thiệu về đường định hướng

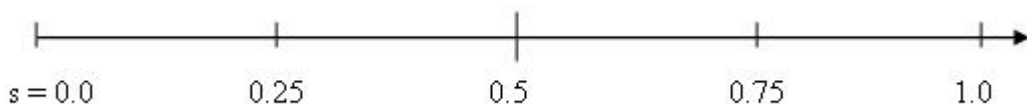
Bên cạnh việc xây dựng diễn hoạt cho bản thân khung xương, trong nhiều tình huống ta cần áp đặt nhân vật di chuyển theo một lộ trình nào đó. Để có thể thực hiện được ta sẽ qui định ra một lộ trình trong đó gồm nhiều con đường kết nối với nhau. Việc sử dụng đường định hướng cho nhân vật là tùy chọn.

5.2.3.2. Cập nhật biến đổi trên các đường cơ bản

Như đã nói, một lộ trình cho nhân vật đi theo sẽ bao gồm trong đó nhiều đường cơ bản. Trong hệ thống diễn hoạt của mình, chúng tôi đã xây dựng các 2 loại đường cơ bản là đường thẳng và đường cong Bezier.

Để nhân vật di chuyển theo đường thì ta sẽ tính vị trí và vector pháp tuyến cho nhân vật vào từng thời điểm dựa theo thời gian đã trôi qua kể từ khi nhân vật di chuyển. Sau khi đã biết vị trí và góc quay hiện tại của nhân vật ta sẽ thiết lập vào giá trị vị trí và góc quay của nút gốc của cấu trúc cây của khung xương.

➤ Đối với đường thẳng:



Hình 5-2 Ví dụ đường đi thẳng

Dựa vào quãng đường đi được của đối tượng từ điểm đầu tiên ta tính được

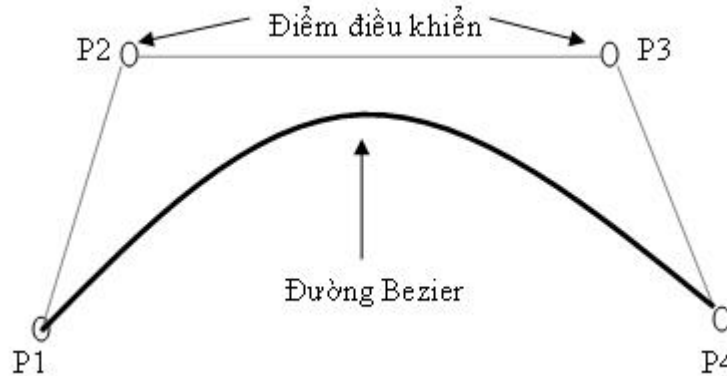
s = khoảng cách / độ dài quãng đường

vị trí hiện tại = vị trí đầu + $s \cdot (\text{vị trí cuối} - \text{vị trí đầu})$

Còn hướng là hướng của đường thẳng (vector cuối - vector đầu)

➤ **Đối với đường cong Bezier:**

Một đường cong Bezier được xác định bằng 4 điểm: điểm đầu, điểm cuối và 2 điểm điều khiển. Ví dụ ta có đường cong Bezier sau:



Hình 5-3 Ví dụ đường đi Bezier

Ở đây chúng tôi không đi sâu vào đường cong Bezier về mặt toán học mà chỉ đề cập đến các công thức dùng trong lập trình. Phương trình toán của đường cong Bezier như sau:

$C(s) = P1*(1-s)^3 + P2*3*s*(1-s)^2 + P3*3*s^2*(1-s) + P4*s^3$ với s là hệ số tỉ lệ:

$s = \text{khoảng cách đến điểm đầu tiên} / \text{khoảng cách cả đường cong}.$

Khoảng cách cả đường cong được xác định gần đúng như sau:

Gọi $length1 = \text{Khoảng cách từ điểm P1 đến P2}.$

$length2 = \text{Khoảng cách từ điểm P2 đến P3}.$

$length3 = \text{Khoảng cách từ điểm P3 đến P4}.$

$length4 = \text{Khoảng cách từ điểm P1 đến P4}.$

$\text{Khoảng cách quãng đường} = (length1 + length2 + length3)*0.5 + length4*0.5$

Giả sử ta đã có 4 vector cho biết vị trí của các điểm là P1, P2, P3, P4 là $vec1$, $vec2$, $vec3$, $vec4$ và ta cần tính vector hiện tại của một điểm cách điểm đầu một khoảng nào đó, ta thực hiện như sau:

Ta tính s dựa trên công thức ở trên.

Gọi $vecOut$ là vector vị trí của điểm hiện tại ta có:

$vecOut = vec1*(1-s)^3 + vec2*3*s*(1-s)^2 + vec3*3*s^2*(1-s) + vec4*s^3.$

Và tiếp tuyến được tính bằng cách lấy đạo hàm của phương trình đường cong.

5.2.4. Vấn đề về quản lý diễn hoạt

5.2.4.1. Các vấn đề cơ bản trong diễn hoạt

➤ Các loại diễn hoạt:

Có hai loại diễn hoạt chính của các đối tượng là diễn hoạt khung xương và diễn hoạt nội suy. Diễn hoạt khung xương tức là các sự chuyển động dựa trên cấu trúc khung xương và thực hiện diễn hoạt dựa vào các khung hình làm khoá. Diễn hoạt nội suy là việc thực hiện diễn hoạt bằng phép nội suy giữa chuyển động ngay trước và ngay sau nó. Trong các Game ngày nay hầu hết đề lưu các mô hình bằng cấu trúc khung xương và kết hợp cả hai loại diễn hoạt để tạo ra một chuyển động đẹp mắt.

➤ Vấn đề thời gian trong diễn hoạt:

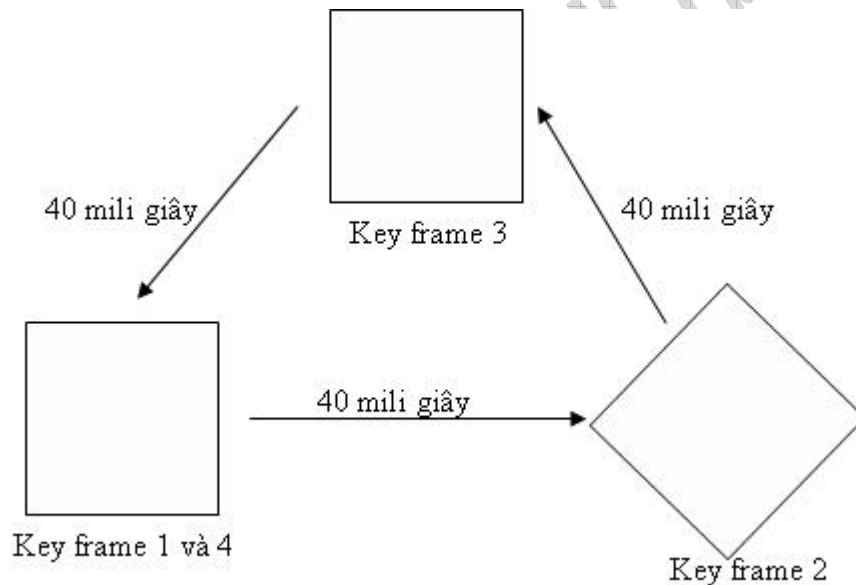
Các Game thực hiện trong một hệ thống thời gian thực. Thời gian là yếu tố quan trọng cần chú ý khi xây dựng Game. Như ta biết trong một chương trình Game thực chất ta có một vòng lặp vô tận, tại mỗi vòng lặp ta thực hiện quá trình hiển thị tương ứng với thời gian đó. Trong hệ thống diễn hoạt thì kết quả cuối cùng là phải có một chuyển động mượt mà và không bị giật do đó phải đảm bảo được số khung hình trong một giây cao hay là thời gian giữa các vòng lặp chính phải ngắn.

Một điều quan trọng trong yếu tố thời gian là ta phải làm sao cho hệ thống diễn hoạt của ta hoạt động giống nhau trên những máy tính có tốc độ xử lý khác nhau. Ví dụ trên một máy tốc độ 2GHz và trên một máy tốc độ 500MHz thì trên máy 2GHz chuyển động sẽ mịn hơn trên máy 500MHz nhưng phải đảm bảo được chuyển động giống nhau vì chúng thực hiện trong thời gian như nhau. Để giải quyết ta sẽ thực hiện bằng cách tính khung hình hiện tại bằng nội suy giữa các khung hình trước và sau nó.

➤ Các khung hình trong chuyển động:

Một chuyển động của mô hình là một dãy các biến đổi của mô hình đó theo thời gian. Ta không thể lưu toàn bộ các biến đổi bởi vì làm như vậy sẽ rất lãng phí bộ nhớ mà thay vào đó ta lưu sự biến đổi của mô hình tại một số mốc nào đó. Ví dụ

như ta lưu sự biến đổi của đối tượng cứ sau vài mili giây chẳng hạn. Các khung hình chính này có thể lưu cách nhau những khoảng thời gian khác nhau hoặc đều nhau đều được. Dưới đây là một ví dụ về chuyển động của vật thể theo key frame với khoảng cách giữa các key frame là 40 mili giây.



Hình 5-4 Ví dụ diễn hoạt qua các khung hình khóa

Như vậy để thực hiện diễn hoạt thì khi chương trình yêu cầu cập nhật thì dựa vào thời gian trôi qua truyền vào ta sẽ tính được frame nào cần hiển thị và sẽ hiển thị key frame đó lên. Nhưng rõ ràng chỉ dựa vào các key frame để hiển thị thì hình ảnh sẽ không mượt mà do sự biến đổi giữa các khung hình là khá lớn. Để khắc phục việc này ta cần kết hợp thêm biến đổi nội suy vào.

➤ **Kết hợp biến đổi nội suy giữa các khung hình:**

Để thực hiện được chuyển động nội suy giữa các key frame ta phải xác định được thời gian đã trôi qua trong chuyển động. Cứ mỗi lần chương trình yêu cầu ta cập nhật và truyền vào đối số là thời gian đã trôi qua thì ta tính toán được thời điểm trong chuyển động vì ta có thời gian tổng cộng của chuyển động và tính được thời gian từ khi chuyển động. Tiếp đó ta sẽ xác định được key frame ngay trước và ngay sau nó. Giả sử ta có được t_1 là thời gian thực hiện key frame n và t_2 là thời gian của key frame $n+1$. Ta tính được hằng số tỉ lệ:

$s = (t-t_1)/(t_2-t_1)$ với t là thời gian hiện thời. Nhờ hệ số này ta dễ dàng nội suy ra biến đổi hiện tại dựa vào key frame n và $n+1$:

Khung hình lúc $t = \text{khung hình lúc } t_1 + s * (\text{khung hình lúc } t_2 - \text{khung hình lúc } t_1)$

Sự biến đổi như vậy có 2 ưu điểm:

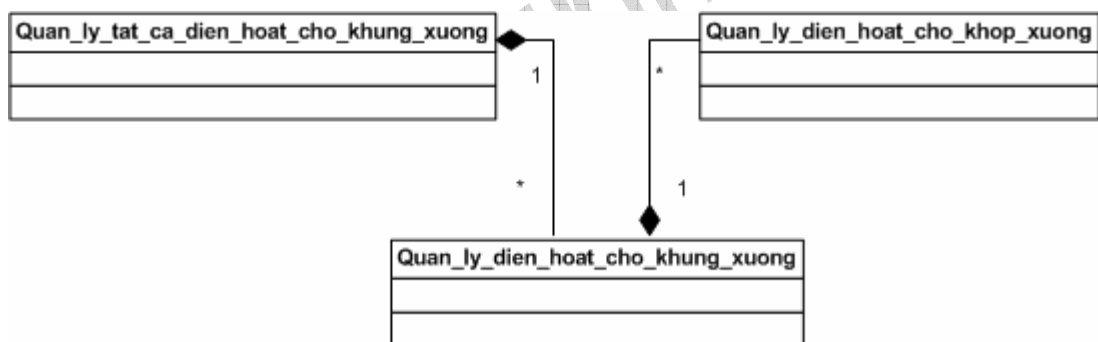
✚ Chuyển động sẽ mượt mà, đẹp và thực hơn.

✚ Chuyển động sẽ diễn ra giống nhau trên những máy có tốc độ CPU khác nhau.

5.2.4.2. Tổ chức quản lý diễn hoạt

Như ta đã biết, một nhân vật có một khung xương và kèm theo là một số các diễn hoạt riêng biệt như đứng, đi, nhảy, tấn công, ... Chúng ta cần hệ thống diễn hoạt cho phép quản lý được tất cả các diễn hoạt cho một nhân vật.

Để thực hiện được yêu cầu đó ta phải tổ chức quản lý từ thấp đến cao, từ chi tiết đến tổng quát. Cụ thể ta xây dựng lớp quản lý một diễn hoạt cho một khớp xương, dựa vào lớp này ta xây dựng lớp quản lý một diễn hoạt cho toàn bộ khung xương, và cuối cùng dựa vào lớp vừa xây dựng ta xây dựng tiếp lớp quản lý tất cả các diễn hoạt cho một khung xương. Tại lớp tổng quát nhất (quản lý toàn bộ diễn hoạt) ta có thể thực hiện các kết hợp các diễn hoạt lại với nhau. Quá trình tích hợp minh họa theo sơ đồ lớp sau:



Hình 5-5 Sơ đồ quan hệ các lớp quản lý diễn hoạt

➤ **Quản lý diễn hoạt cho một khớp xương:**

Trong lớp quản lý diễn hoạt cho một khớp xương ta cần có các thông tin như sau:

✚ Con trỏ chỉ đến khớp xương trong khung xương mà ta cần quản lý. Tất cả những thông tin về khớp xương sẽ được ta truy cập dựa vào con trỏ này.

✚ Phải lưu các thông tin diễn hoạt của một khớp xương. Thông tin này là một cấu trúc lưu thông tin diễn hoạt được nạp từ tập tin về chính khớp xương, về khớp xương cha, về danh sách các khung hình của diễn hoạt.

Trong lớp quản lý này ta cần phải cung cấp các phương thức cho phép truy xuất đến vector vị trí và quaternion quay của các khung hình trong chuỗi các khung hình biến đổi. Chúng ta còn có thể thực hiện cập nhật lại trạng thái cho khớp xương khi biết khớp xương đang ở vào thời điểm nào của diễn hoạt.

➤ **Quản lý diễn hoạt cho toàn bộ khung xương:**

Việc quản lý cho toàn bộ khung xương cần có một danh sách các lớp quản lý cho một khớp xương. Trong lớp quản lý này ta cần quản lý các thông tin liên quan đến diễn hoạt chung cho khung xương như: khoảng thời gian trôi qua từ khi diễn hoạt từ khung hình thứ nhất, diễn hoạt có lặp đi lặp lại không, khoảng cách di chuyển của khung xương kể từ lần cập nhật trước và tổng khoảng cách di chuyển của khung xương trong toàn bộ diễn hoạt.

Khi cần cập nhật diễn hoạt cho toàn bộ khung xương ta sẽ thực hiện cập nhật diễn hoạt cho từng khớp xương ở trong khung xương. Trong lớp này ta cần phải cung cấp các phương thức cho phép hiệu chỉnh thay đổi diễn hoạt như về thời gian diễn hoạt, khoảng cách trong diễn hoạt.

➤ **Quản lý tất cả diễn hoạt cho khung xương:**

Một khung xương có nhiều diễn hoạt, ta cần danh sách các đối tượng với mỗi đối tượng quản lý cho một diễn hoạt. Thông thường tại một thời điểm ta có một

diễn hoạt được kích hoạt. Tại lớp này ta có thể thực hiện các chức năng nâng cao như kết hợp các diễn hoạt vào một khung xương.

5.2.5. Kết hợp các diễn hoạt

5.2.5.1. Kết hợp các diễn hoạt khác nhau vào khung xương

➤ **Giới thiệu:**

Đây là khả năng cho phép ta kết hợp nhiều chuyển động riêng biệt lại thành 1 chuyển động kết hợp.

➤ **Lý do xây dựng:**

Đối với một mô hình ví dụ như một nhân vật trong Game do nhiều yếu tố ràng buộc như thời gian thực hiện và không gian lưu trữ nên ta chỉ xây dựng cho nhân vật đó một số các diễn hoạt cơ bản mà thôi. Ví dụ như ta có một quái vật thì ta chỉ có thể tạo cho chúng các diễn hoạt cơ bản như đứng, đi, chạy, tấn công, chết, lách qua trái, lách qua phải. Nhưng trong Game đôi khi ta có nhu cầu nhân vật thực hiện các diễn hoạt phức tạp là sự kết hợp của nhiều diễn hoạt riêng rẽ ví dụ như nhân vật vừa đi vừa bắn súng được kết hợp từ hai diễn hoạt riêng biệt là đi và bắn súng. Từ nhu cầu như vậy Game Engine phải có chức năng cho phép người lập trình thực hiện việc kết hợp các biến đổi này. Với chức năng kết hợp các diễn hoạt thì từ một tập các diễn hoạt nhỏ ta có thể tạo ra một tổ hợp nhiều diễn hoạt khác nhau.

➤ **Cách thức thực hiện:**

Một chuyển động của khung xương đơn thuần chỉ là các ma trận biến đổi cho các khớp của khung xương. Sự biến đổi gồm có sự dịch chuyển, sự quay và biến đổi tỉ lệ. Một xương thì chuyển động quay xung quanh một khớp, chỉ có khớp gốc là chuyển động tương đối so với thế giới mà thôi.

Để tạo được sự kết hợp chuyển động thì ta sẽ tạo ra sự kết hợp giữa các ma trận khoá của nhiều chuyển động. Chú ý rằng ma trận kết hợp là không thể giao hoán được có nghĩa là thứ tự kết hợp là quan trọng. Nếu muốn nhân 2 sự biến đổi có sự

quay và chuyển dịch thì bạn phải kết thúc bằng một biến đổi cuối theo thứ tự: quay, chuyển dịch, quay và cuối cùng là chuyển dịch.

Để thực hiện đúng thì chúng ta phải cộng các sự biến đổi thay vì nhân. Ví dụ như có 2 biến đổi là quay và chuyển dịch cần phải kết hợp vào một chuyển động thì việc cộng các biến đổi là hợp lý.

Tóm lại, khi thực hiện kết hợp các chuyển động ta cần biết các thông tin về tỉ lệ kết hợp của các chuyển động vào chuyển động chung. Khi yêu cầu cập nhật lại diễn hoạt ta sẽ tính toán ra các ma trận nội suy cho từng diễn hoạt tại thời điểm đó. Sau khi đã có các ma trận nội suy của từng diễn hoạt ta sẽ thực hiện cộng tất cả ma trận nội suy này lại với tỉ lệ đóng góp vào chuyển động đã được biết.

5.2.5.2. Kết hợp các diễn hoạt trong các phần của khung xương

➤ Giới thiệu:

Đôi khi trong một hệ thống khung xương không chỉ có một chuyển động duy nhất mà có nhiều chuyển động trong đó. Ví dụ như phần khung xương của một thân thể, có thể phần dưới (bụng đến chân) thực hiện diễn hoạt chạy, phần thân trên (từ bụng lên cổ) lại thực hiện diễn hoạt bắn súng. Như vậy giữa 2 phần thân trên và thân dưới thực hiện 2 diễn hoạt khác nhau. Phần diễn hoạt phần thân dưới phải có tác động nhất định lên phần thân trên và ngược lại. Nếu như chúng ta không thực hiện kết hợp hay điều hòa diễn hoạt tại bụng (nơi giao nhau giữa 2 diễn hoạt) thì nhân vật sẽ chuyển động rất không tự nhiên. Kết hợp diễn hoạt trong các phần khung xương chính là kết hợp các diễn hoạt khác nhau trong một khung xương để tạo ra hiệu ứng tự nhiên.

➤ Lý do xây dựng:

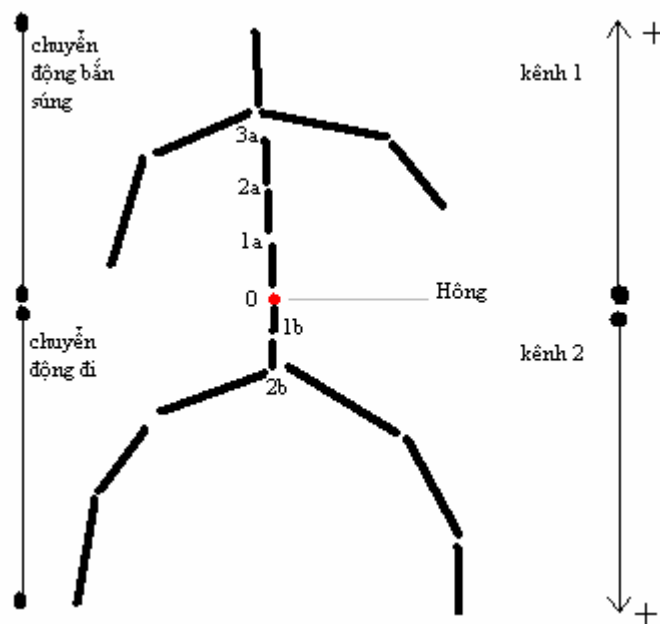
Việc xây dựng chức năng phối hợp diễn hoạt giữa các thành phần trong cùng khung xương sẽ giải cho phép ta thực hiện nhiều diễn hoạt cho các phần trong cùng khung xương mà vẫn đảm bảo tính tự nhiên.

Việc xây dựng này còn giúp ta tiết kiệm rất nhiều diễn hoạt vì ta có thể thực hiện kết hợp các diễn hoạt ngay trong khung xương. Ví dụ như trong phần thân nhân vật ta có thể tạo ra chuyển động bắn kết hợp với đi, chạy, nhảy cần 4 diễn hoạt. Nếu ta không thực hiện kết hợp ta cần xây dựng thêm các diễn hoạt đi bắn, chạy bắn, nhảy bắn. Rõ ràng việc phối hợp các chuyển động giúp ta giảm nhẹ công việc đi rất nhiều và tăng hiệu quả thực hiện lên rất cao.

➤ **Cách thức thực hiện:**

Để tiện minh họa cách thực hiện ta xét ví dụ cụ thể sau:

Ta có một khung xương là thân người với khớp đầu tiên (chỉ số là 0) là khớp 1b, 2 diễn hoạt là đi và bắn súng. Ta cần thực hiện kết hợp 2 chuyển động này vào 2 phần khác nhau trong cấu trúc khung xương. Phần từ hông trở xuống ta thực hiện diễn hoạt đi, còn phần từ hông trở lên ta thực hiện diễn hoạt bắn. Hình khung xương như sau:



Hình 5-6 Minh họa kết hợp chuyển động các phần trong khung xương

Trước hết ta sẽ chia phần khung xương thân này ra làm 2 kênh. Kênh thứ nhất từ khớp 2a trở đi đến hết. Kênh thứ 2 từ khớp 1b đến khớp 1a. Ở đây một kênh là một

khái niệm ta đưa ra để quản lý việc kết hợp các diễn hoạt. Một kênh sẽ gồm 4 thông số sau:

✚ Khớp bắt đầu của kênh.

✚ Khớp kết thúc của kênh. Chỉ số của khớp kết thúc phải lớn hơn hay bằng chỉ số của khớp bắt đầu.

✚ Chỉ số fade-in: chỉ ra số khớp xương bên trong kênh chịu ảnh hưởng bởi diễn hoạt. Ví dụ nếu kênh thứ nhất có chỉ số fade-in = 2 thì có nghĩa rằng 2 khớp 2a và khớp 3a sẽ bị ảnh hưởng khi ta thực hiện kết hợp diễn hoạt. Cụ thể 2 khớp 2a và khớp tiếp theo 2a bị tác động của chuyển động bắn súng. Tỷ lệ ảnh hưởng của diễn hoạt bắn súng lên khớp sẽ tăng dần theo chỉ số. Công thức để tính tỷ lệ ảnh hưởng như sau:

$$\text{fadeInWeight} = 1 / (\text{fadeIn} + 1)$$

$$\text{Tỷ lệ cho khớp có chỉ số } i = \text{fadeInWeight} * (i - \text{chỉ số khớp đầu của kênh} + 1)$$

✚ Chỉ số fade-out: chỉ số này chỉ ra số khớp xương bên ngoài kênh kể từ kênh cuối chịu ảnh hưởng bởi diễn hoạt. Ví dụ như ta thiết lập chỉ số fade-out của kênh thứ 2 là 2 thì 2 khớp xương 2a và 3a sẽ bị tác động. Tỷ lệ tác động của diễn hoạt (cụ thể là đi) sẽ giảm dần theo chiều tăng của chỉ số. Công thức để tính tỷ lệ ảnh hưởng như sau:

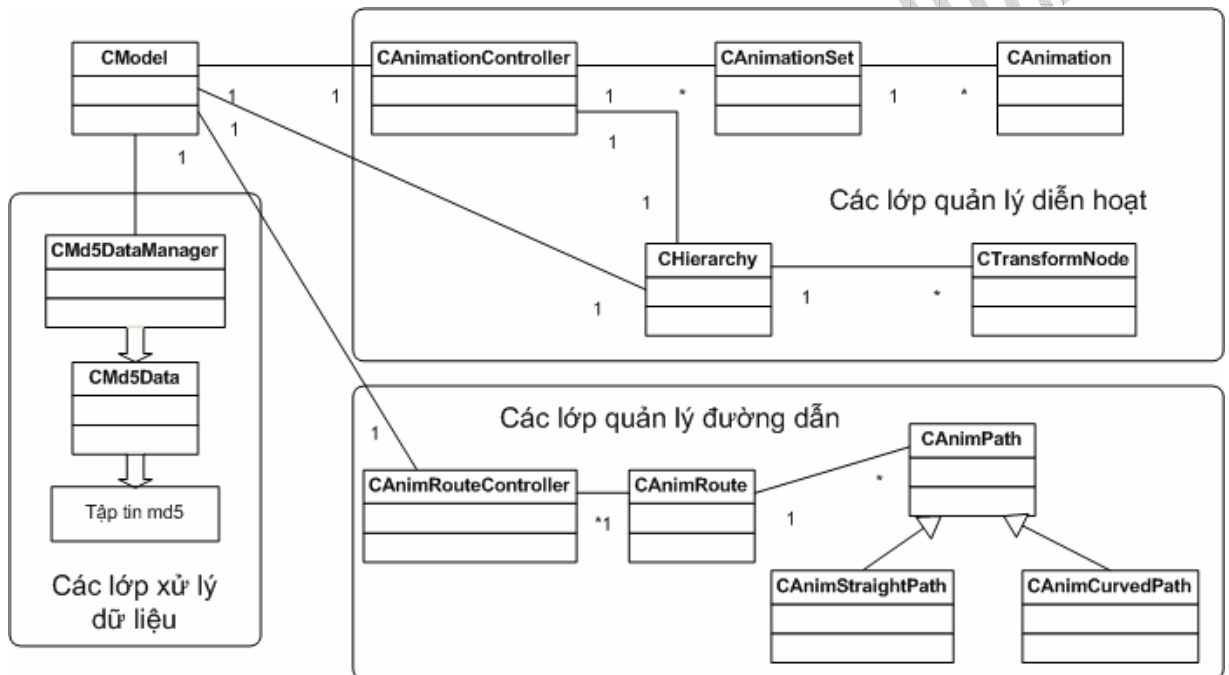
$$\text{fadeOutWeight} = 1 / (\text{fadeOut} + 1)$$

$$\text{Tỷ lệ cho khớp có chỉ số } i = 1 - (\text{fadeOutWeight} * (i - \text{chỉ số khớp cuối}))$$

Để chuyển động được chính xác ta phải đảm bảo tổng tỷ lệ tác động lên khớp xương phải bằng 1. Ví dụ với khớp xương 2a bị tác động của kênh một tỷ lệ là $1/(\text{fade-in}+1) = 1/3$ thì tỷ lệ của kênh 2 tác động lên khớp 2a phải là $2/3$. Để có thể đạt được sự cân bằng như vậy thì chỉ số fade-in của kênh một phải bằng chỉ số fade-out của kênh 2 và ngược lại.

5.3. Hệ thống diễn hoạt trong thực thi

5.3.1. Sơ đồ lớp của hệ thống diễn hoạt



Hình 5-7 Sơ đồ lớp của hệ thống diễn hoạt

5.3.2. Chức năng các thành phần trong sơ đồ

Một đối tượng thuộc lớp Model sẽ cần thông tin liên quan đến diễn hoạt và đường dẫn. Các chức năng hệ thống được chia ra 3 nhóm:

5.3.2.1. Hệ thống xử lý dữ liệu

Đây chính là thành phần xử lý dữ liệu về diễn hoạt được lưu trên tập tin định dạng **md5**. Cụ thể các lớp như sau:

🔧 Lớp `CMd5Data`: Cung cấp các hàm đọc dữ liệu từ các tập tin vào trong các cấu trúc dữ liệu định sẵn.

🔧 Lớp `CMd5DataManager`: Thực hiện quản lý việc đọc dữ liệu từ tập tin md5. Nếu có yêu cầu truy xuất đến dữ liệu và dữ liệu chưa được đọc thì ta sẽ thực hiện đọc dữ liệu lên, còn ngược lại ta chỉ cần trả về và xử dụng dữ liệu trong đối tượng `CMd5Data` đã có.

Mọi dữ liệu về diễn hoạt cần thiết thì các đối tượng khác phải truy xuất thông qua hệ thống xử lý này.

5.3.2.2. Các lớp quản lý đường dẫn

Đây là các lớp thực hiện quản lý đường định hướng di chuyển cho nhân vật. Cụ thể có các lớp sau:

✚ Lớp CAnimPath: là lớp quản lý thông tin cho một đường trong lộ trình. Các thông tin nó quản lý bao gồm điểm đầu, điểm cuối, và độ dài của lộ trình.

✚ Lớp CAnimStraightPath: được kế thừa từ lớp CAnimPath để cài đặt cụ thể cách tính tọa độ và vector pháp tuyến cho nhân vật trên đường thẳng khi biết nhân vật cách điểm xuất phát một khoảng cách nào đó.

✚ Lớp CAnimCurvedPath: được kế thừa từ lớp CAnimPath để cài đặt cụ thể cách tính tọa độ và vector pháp tuyến cho nhân vật trên đường cong Bezier khi biết nhân vật cách điểm xuất phát một khoảng cách nào đó.

✚ Lớp CAnimRoute: lớp này quản lý thông tin cho một lộ trình. Một lộ trình là sự kết hợp nhiều đường lại với nhau. Lớp này sẽ tính toán vị trí và vector pháp tuyến cho đối tượng khi biết khoảng cách từ đối tượng đến điểm xuất phát trên lộ trình.

✚ Lớp CAnimRouteController: Quản lý thông tin tất cả lộ trình cho nhân vật. Trong một thời điểm chỉ có nhiều nhất một lộ trình được sử dụng.

5.3.2.3. Các lớp quản lý diễn hoạt

Các lớp này là cài đặt cụ thể của các lớp quản lý diễn hoạt đã trình bày trong phần trước.

✚ Lớp CTransformNode: Lớp này định nghĩa một đối tượng lưu các thông tin biến đổi. Các thông tin của lớp quản lý bao gồm vị trí và góc quay của một đối tượng trong mối tương quan với đối tượng cha và với thế giới.

✚ Lớp CHierarchy: Quản lý cấu trúc khung xương của nhân vật. Mỗi một khớp trong khung xương là một đối tượng thuộc lớp được kế thừa từ lớp CTransformNode.

🚦 Lớp CAnimation: Quản lý thông tin và thực hiện diễn hoạt cho một khớp của cấu trúc khung xương. Lớp này sẽ chịu trách nhiệm cập nhật lại vị trí và góc quay cho khớp xương trong diễn hoạt.

🚦 Lớp CAnimationSet: Quản lý thông tin và thực hiện diễn hoạt cho tất cả các khớp trong khung xương. Thật sự lớp này sẽ bao gồm tập các đối tượng thuộc lớp CAnimation. Lớp này phải quản lý được thông tin chung cho một diễn hoạt.

🚦 Lớp CAnimationController: Đây là lớp bao bọc quản lý tất cả thông tin diễn hoạt cho một đối tượng. Trong lớp này ta thực hiện khởi tạo dữ liệu, chọn diễn hoạt, phân thành các kênh diễn hoạt, thực hiện và hiệu chỉnh thông tin diễn hoạt, và phối hợp các diễn hoạt khác nhau trên cùng một đối tượng.

5.4. Tóm tắt

Hệ thống diễn hoạt là một thành phần hết sức quan trọng trong việc xây dựng Game. Dựa trên đặc điểm và yêu cầu của một hệ thống diễn hoạt và việc tham khảo nhiều Game Engine như Half-live, Cal3D và NeoEngine, chúng tôi đã phân tích, thiết kế và xây dựng lên một hệ thống đảm bảo được tốc độ, tính trong sáng dễ sử dụng, khả năng mở rộng cao, và cung cấp ra các chức năng nâng cao cho phép giải quyết được những tình huống diễn hoạt trong Game.

Chương 6 Hệ thống vật lý (Physics System)

- ◇ [Giới thiệu hệ thống vật lý](#)
- ◇ [Các yếu tố cần xử lý trong hệ thống vật lý](#)
- ◇ [Engine vật lý NovodeX](#)
- ◇ [Sử dụng NovodeX](#)
- ◇ [Tóm tắt](#)

6.1. Giới thiệu hệ thống vật lý

Hiện nay đòi hỏi của người chơi về chất lượng Game rất cao. Bên cạnh Game phải có hình ảnh đẹp, nội dung hay còn yêu cầu phải chân thực, gần với thực tế bên ngoài. Để cho Game mô phỏng được các tình huống xảy ra trong thực tế thì phải xây dựng một hệ thống quản lý các tương tác vật lý.

Có thể nói rằng tương lai của Game chính là vật lý. Game sẽ trở nên rất thật khi xử lý tốt được những hiệu ứng theo quy luật vật lý như va chạm, nổ, chuyển động của các vật trong cảnh Game, nhân vật di chuyển và tương tác với các yếu tố trong môi trường, và các hiệu ứng đặc biệt như mô phỏng quần áo, tóc bay trong gió.

Tóm lại, hệ thống vật lý giúp chúng ta có thể xử lý cách các đối tượng di chuyển, cách chúng ứng xử trong môi trường với từng tình huống cụ thể, và cách chúng tương tác với nhau. Nhờ hệ thống vật lý mà Game của chúng ta sẽ sinh động, tự nhiên.

6.2. Các yếu tố cần xử lý trong hệ thống vật lý

➤ Các thuộc tính của vật liệu

Các thuộc tính của vật liệu chính là các đặc trưng vật lý về vật chất như trọng lượng, ma sát, đàn hồi. Chúng ta có thể tạo ra các mặt phẳng trơn khiến cho nhân vật có khó di chuyển được, các vật được làm bằng chất liệu gỗ với các chỗ nối có thể gãy được khi có lực va chạm lớn, các bề mặt đàn hồi như cao su, các bề mặt bằng kim loại có thể uốn cong được khi tác động lực, và có thể tạo được các tảng đá có thể vỡ vụn khi bị tác động lực mạnh.

➤ Các đối tượng bao bọc động và phát hiện va chạm

Các đối tượng bao bọc giúp chúng ta bao các đối tượng trong Game để thực hiện các hiệu ứng vật lý lên đối tượng đó và phát sinh ra các sự kiện tương ứng khi các va chạm xảy ra giúp chúng ta có thể đưa ra các cách thức xử lý tương ứng. Để thực hiện được sự mô phỏng vật lý như vậy chúng ta phải dựa vào các định luật vật lý cổ điển trong đó quan nhất là các định luật Newton. Các định luật vật lý này giúp

chúng ta có thể tạo được các va chạm tự nhiên và nhiều hiệu ứng đẹp mắt ví dụ như sự rung khi một đồng tiền rơi xuống sàn nhà, chiếc xe trượt theo quán tính khi phanh, ... Hầu hết các Engine vật lý xây dựng nên các hình bao bọc động như là những yếu tố cốt lõi và xây dựng các yếu tố khác thông qua chúng.

➤ **Các khớp và sự co giãn**

Các khớp và các đặc tính co giãn dùng để mô hình hoá cho các cơ cấu phức tạp như các máy móc, xe cộ, sự di chuyển của các nhân vật, cửa, đòn bẩy, và cung cấp khả năng cho các nhân vật cầm và thao tác các đối tượng trong Game.

➤ **Các chất lỏng và vô định hình**

Bên cạnh việc mô phỏng các hiệu ứng như ngọn sóng trên mặt hồ thì chúng ta còn có thể tăng thêm hiệu ứng chất lỏng trong nhiều cảnh vật như dầu, các cột nước bắn lên, lửa, các vũ khí bắn ra các chất vô định hình. Các chất lỏng này có thể tương tác với các vật động trong cảnh như bao quanh chúng chẳng hạn.

➤ **Áo quần và hiệu ứng khác**

Hệ thống vật lý có thể giúp mô phỏng chính xác, trung thực về áo quần như áo choàng cho nhân vật. Hay hệ thống có thể cung cấp cho ta chức năng mô phỏng về lửa, khói, sương mù. Tuy nhiên đây chỉ là một khía cạnh phụ có thể bỏ qua trong hệ thống vật lý, ta có thể xây dựng hệ thống Particle riêng để thực hiện cho các hiệu ứng này.

6.3. Engine vật lý NovodeX

Hiện nay có rất nhiều Engine vật lý được tạo ra để kết hợp vào các 3D Engine để tạo ra các Game. Một số Engine vật lý tốt có thể kể đến là Navok được dùng trong Game Haff Life và NovodeX.

NovodeX là Engine vật lý của hãng AGEIA. Hiện nay (tháng 6 năm 2005) NovodeX đang có phiên bản 2.2 và chưa được hoàn thiện. NovodeX là một Engine vật lý rất mạnh và cung cấp miễn phí cho các hoạt động phi thương mại. Engine vật lý này còn có kèm nhiều ví dụ và tài liệu hướng dẫn cũng chính vì vậy chúng tôi

chọn để nghiên cứu và ứng dụng kết hợp với Nwfc 3D Engine để xây dựng Game ứng dụng.

Engine vật lý NovodeX cung cấp cho chúng ta một giải pháp rất mạnh và hiệu quả để kết hợp một hệ thống xử lý vật lý thời gian thực vào trong hệ thống Game của chúng ta. NovodeX được thiết kế để dễ dàng kết hợp với các thành phần khác trong một Game Engine. NovodeX có thể làm việc tốt trên nhiều API khác nhau như Direct3D, OpenLG, NDL Gamebryo, Quake và OGRE. NovodeX còn thể thể được bao bọc bởi Unreal 3 của Engine Epic Game và Reality Engine của Artificial Studio. Những kết quả đầu ra của NovodeX còn có thể cung cấp cho Discreet 3D Studio Max, SoftImage XSI và các công cụ khác cho phép các nhà phát triển có thể xem công việc của mình thực hiện trong những môi trường tương tự.

Để dễ dàng kết hợp với những công nghệ kèm theo, bản thân NovodeX cung cấp một loạt các công cụ để tạo ra các hiệu ứng và môi trường động dựa trên vật lý. Những công cụ đó bao gồm:

- ✚ Rocket viewer để cài đặt và mô phỏng cho các cảnh vật lý.
- ✚ Debug render cho phép chúng ta phân tích và chỉnh lại Game.
- ✚ Bộ xử lý dữ liệu cho phép ta nạp dữ liệu vào nhanh chóng.

NovodeX được thiết kế hoạt động tối ưu trên trên chip AGEIA PhysX. NovodeX là API duy nhất cho phép các nhà phát triển Game dùng phần cứng để gia tăng các hiệu ứng vật lý trong Game dựa trên AGEIS PhysX chip, đây là đơn vị xử lý vật lý (PPU – Physics Processing Unit) đầu tiên trên thế giới. NovodeX còn là API đa tiểu trình đầu tiên và duy nhất hiện nay cho phép các nhà phát triển Game sử dụng để tăng thêm khả năng xử lý trong Game của mình.

NovodeX cung cấp hệ thống hoàn hảo các đối tượng bao bọc động và hệ thống phát hiện, xử lý va chạm tốt. NovodeX có thể thực thi trên nhiều môi trường khác nhau như Microsoft Window XP, Mac OS X, PhysX PPU. Một điểm mạnh nữa của NovodeX là nó cung cấp sẵn cho chúng ta các lớp dùng để điều khiển di chuyển các nhân vật.

➤ **Các chức năng của NovodeX trong lúc thực thi chương trình gồm:**

✚ Hệ thống kiểm soát độ cong cho mặt phẳng, hộp, khối cầu, đỉnh các vật thể, mesh của môi trường, các điểm lõm của đối tượng.

✚ Cho phép tạo ra các nhóm va chạm. Tức là chúng ta có thể nhóm 1 số đối tượng thành một nhóm để tiện cho việc kiểm tra và xử lý va chạm giữa các đối tượng này.

✚ Cung cấp các loại khớp như Fixed (cố định), Revolute (bọc ở bên ngoài theo một trục), Spherical (cầu), Prismatic (lăng trụ), và Six Degree of Freedom (6 bậc tự do) (tham khảo thêm ở phần phụ lục) cùng với các phép chiếu, độ đàn hồi, qui định giới hạn cho các khớp quay.

✚ Các vật liệu cho các bề mặt với các hệ số ma sát động, ma sát tĩnh và độ đàn hồi.

✚ Cung cấp những hình trigger cho phép chúng ta có thể xử lý các sự kiện khi có một vật bắt đầu va chạm, đi vào hay ra khỏi một vật thể nào đó.

✚ Cung cấp tia chiếu từ một vật thể vào môi trường xung quanh. Nhờ tia chiếu này chúng ta có thể xác định được khoảng cách cũng như tính chất của đối tượng có cắt với tia chiếu và có thể có những ứng xử tương ứng. Đây cũng là cách chúng ta có thể xây dựng những ứng xử thông minh đơn giản cho các đối tượng, ví dụ khi con quái vật đến gần nhân vật một khoảng nào đó thì ta cho con quái vật tấn công.

✚ Cung cấp các lớp do người dùng định nghĩa với các hàm xử lý các sự kiện trả về. Đây là những hàm rất quan trọng cho phép chúng ta có thể xử lý khi có va chạm xảy ra. Ví dụ khi nhân vật di chuyển trong môi trường và va chạm một hộp thì sẽ có sự kiện va chạm trả về, trong hàm xử lý va chạm ta có thể thiết lập lực tác động vào cái hộp chẳng hạn.

✚ Cung cấp một bộ điều khiển nhân vật dựa trên một chiếc hộp tạm. Khi di chuyển nhân vật, chiếc hộp tạm này thực hiện di chuyển trước để xác định xem

nhân vật sẽ phải di chuyển tiếp theo như thế nào để có thể áp dụng lên nhân vật cho chính xác.

- Thực hiện đồng bộ hoá hệ thống vật lý với một bộ xử lý đa tiêu trình.

- Cho phép hiển thị chế độ Debug cho hệ thống tọa độ, các đối tượng có trong cảnh, các điểm va chạm, các vector pháp tuyến, ...

➤ **Các công cụ phát triển kèm theo gồm có:**

- Xuất ra định dạng tập tin PML có thể sử dụng trong các phần mềm 3DS Max, SoftImage, và Maya.

- Cung cấp sẵn mã nguồn cho việc đưa các đối tượng trong định dạng tập tin PML vào trong cảnh của NovodeX.

- NovodeX Rocket, một ứng dụng riêng biệt cho phép ta nạp và mô phỏng một cảnh với định dạng PML.

- NovodeX FX, một môi trường kết hợp để mô phỏng các hiệu ứng nâng cao trong các hệ thống quản lý Game hiện có.

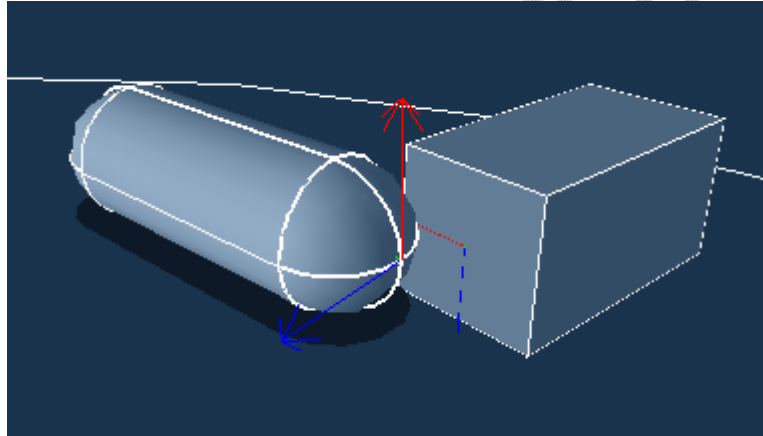
6.4. Sử dụng NovodeX

6.4.1. Kết hợp NovodeX vào Game

NovodeX là một phần hoàn toàn độc lập, chức năng chính là chỉ quản lý về vật lý và nó được xây dựng trên một thư viện toán học riêng. Do đó để có thể sử dụng được các tính năng của NovodeX thì cần phải bao bọc nó lại để có thể sử dụng kết hợp với các thành phần khác như hệ thống hiển thị, hệ thống diễn hoạt. Việc bao bọc hệ thống vật lý còn giúp chúng ta có thể sử dụng nó một cách thống nhất với các phần khác thông qua cùng một thư viện toán.

Tư tưởng chính để sử dụng hệ thống vật lý NovodeX là ta dùng các vật thể và mesh do NovodeX cung cấp để bao bọc hay chứa các vật thể và nhân vật của Game. Việc quản lý vật lý ta để cho hệ thống NovodeX thực hiện. Sau khi hệ thống NovodeX thực hiện xong ta lấy vị trí và góc quay mới của các vật thể bao bọc của NovodeX để cập nhật lại vị trí và góc quay tương ứng cho các vật thể và nhân vật trong Game. Ví dụ như ta có một khối hộp trong màn Game, ta sẽ dùng một đối

tượng hộp cùng kích của NovodeX để bao bọc nó lại. Việc xử lý va chạm hay việc mô phỏng chuyển động cho khối hộp là do hệ thống NovodeX quản lý thông qua khối hộp bao đối tượng. Sau khi hệ thống NovodeX xử lý rồi thì ta lấy sự biến đổi của khối hộp bao bọc để cập nhật lại trạng thái cho khối hộp trong Game.



Hình 6-1 Ví dụ bao bọc đối tượng Game bằng đối tượng của NovodeX

Trong ví dụ ở hình vẽ trên, ta có hai đối tượng một là một khối hình chữ nhật, một là một khối capsule (hình trụ với hai đầu là hai bán cầu). Ta sẽ dùng hai hình hộp và capsule tương ứng bao bọc lại với cùng kích thước và khớp với vật thể của Game. Bằng cách thực hiện như vậy ta tận dụng được tất cả các tính năng vật lý do NovodeX cung cấp và tất nhiên ta không hiển thị các hình bao bọc trong Game mà chỉ dùng chúng như vật trung gian để xử lý vật lý.

Để thực hiện việc bao bọc các thành phần trong Game bằng các đối tượng vật lý thì bước đầu tiên và quan trọng nhất là ta phải khởi tạo các đối tượng bao bọc. Để làm được việc khởi tạo đó thì sau khi đã xây dựng màn, ta đã xác định được vị trí, kích thước, góc xoay của các đối tượng trong màn Game thì ta tạo ra các vật thể bao bọc tương ứng dựa trên các thuộc tính đó.

NovodeX cung cấp cho ta rất nhiều chức năng để ta có thể hoàn toàn xây dựng một Game chỉ dựa vào nó mà thôi. Tuy nhiên trong Game ta chỉ cần sử dụng một số tính năng chính để xử lý vật lý. Các chức năng về vật lý chính cần dùng là quản lý vật lý cho các đối tượng, quản lý các va chạm giữa các vật thể trong Game, điều khiển nhân vật di chuyển trong màn Game, xác định điểm va chạm dựa vào tia

chiều trong việc xử lý bắn đạn, kết nối các đối tượng thông qua các khớp để tạo nên như vật thể có cơ cấu phức tạp như xe, hệ thống truyền lực, ... Như vậy ta có thể chia hệ thống vật lý ra thành ba thành phần chính sau:

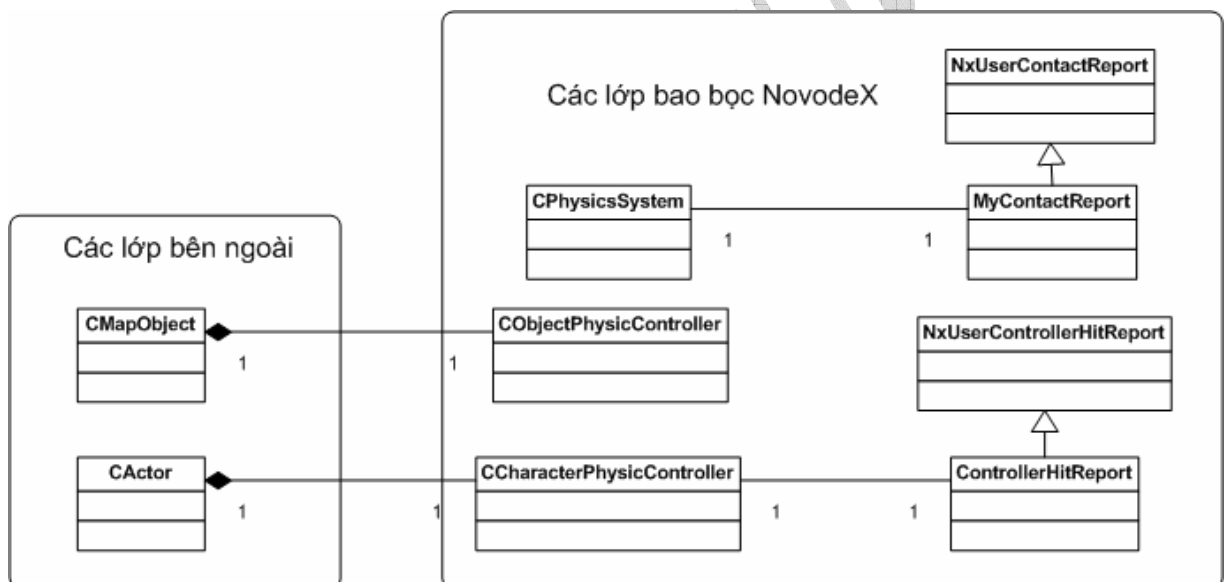
✚ **Thành phần quản lý chung:** đây là thành phần giúp thực hiện việc khởi tạo, huỷ hệ thống cũng như thực hiện thiết lập các thông số đặc trưng cho hệ thống vật lý như gia tốc trọng trường, các loại vật liệu bề mặt, xử lý các sự kiện trả về khi có va chạm xảy ra, thực hiện việc giả lập vật lý, và hiển thị chế độ debug để kiểm tra. Trong Game chỉ tồn tại một đối tượng hệ thống vật lý này mà thôi hay đối tượng vật lý là đối tượng toàn cục.

✚ **Quản lý các đối tượng:** Thành phần này thực hiện quản lý cho các đối tượng tĩnh, động trong màn Game. Thành phần này giúp ta cho khởi tạo các hình bao bọc các đối tượng, thực hiện giả lập vật lý, trả về lại vị trí và góc xoay để ta thiết lập lại thuộc tính cho các đối tượng trong Game.

✚ **Quản lý nhân vật:** Thành phần này giúp ta quản lý sự di chuyển hay xử lý các tình huống xảy ra cho nhân vật như va chạm, tiến đến gần một đối tượng khác.

6.4.2. Cài đặt NovodeX trong ứng dụng

Tương ứng với ba thành phần trên, trong hệ thống vật lý có các lớp sau:



Hình 6-2 Các lớp chính trong hệ thống vật lý

6.4.3. Các thành phần trong sơ đồ

Lớp MyContactReport:

Lớp MyContactReport được kế thừa từ lớp NxUserContactReport đây là lớp của NovodeX cung cấp để người dùng có thể bắt và xử lý các sự kiện khi có va chạm giữa các actor trong hệ thống vật lý. Trong lớp MyContactReport ta sẽ cài đặt hàm ảo onContactNotify để xử lý va chạm cho các đối tượng (trừ các nhân vật).

Lớp ControllerHitReport:

Lớp ControllerHitReport được kế thừa từ lớp NxUserControllerHitReport đây là một lớp cũng do NovodeX cung cấp để xử lý các sự kiện va chạm khi nhân vật của chúng ta va chạm phải một vật cản trong môi trường. Chúng ta sẽ xử lý cho từng nhân vật riêng biệt như thiết lập lực tác động lên vật bị va chạm, xử lý AI đơn giản.

Lớp CPhysicsSystem:

Đây là lớp quản lý toàn bộ hệ thống vật lý. Trong Game sẽ có duy nhất một đối tượng thuộc lớp CPhysicsSystem cho phép ta khởi tạo hệ thống, hủy hệ thống, khởi tạo các thuộc tính chung cho các đối tượng vật lý, xử lý va chạm.

Các loại va chạm mà đối hệ thống vật lý xây dựng có thể xử lý:

```
enum ECOLLISION_TYPE
{
    COLLISION_NONE,
    COLLISION_BOX,
    COLLISION_SPHERE,
    COLLISION_CAPSULE,
    COLLISION_CONVEX,
    COLLISION_MESH,
};
```

Ta sẽ xử lý va chạm cho các hộp, hình cầu, hình capsule (hình trụ có 2 đầu là 2 bán cầu), các mặt lồi được dùng cho các mesh tĩnh và động có số lượng tam giác cầu thành < 256, và cho các mesh tĩnh lớn. Hệ thống vật lý cũng đồng thời quản lý 3 nhóm va chạm:

```
enum EGAME_GROUP
{
    GROUP_NON_COLLIDABLE,
    GROUP_COLLIDABLE_NON_PUSHABLE,
    GROUP_COLLIDABLE_PUSHABLE,
};
```

Với GROUP_NON_COLLIDABLE là nhóm các actor hoàn toàn không thực hiện va chạm với các đối tượng khác hoặc không cần xử lý va chạm cho chúng. GROUP_COLLIDABLE_NON_PUSHABLE là nó có va chạm nhưng ta không thể di chuyển chúng được, và loại cuối cùng là GROUP_COLLIDABLE_PUSHABLE cho phép ta xử lý va chạm và tác động lực để di chuyển chúng.

Một chức năng quan trọng của lớp CPhysicsSystem là cho phép ta tạo ra các vật liệu tương ứng với các bề mặt của các vật có trong Game. Hệ thống còn cho phép tạo lập tốt các hình bao bọc dựa trên thuộc tính của các đối tượng trong Game.

Lớp CObjectPhysicController:

Như trong sơ đồ ta thấy mỗi một đối tượng thuộc lớp CMapObject sẽ chứa một đối tượng thuộc lớp CObjectPhysicController. Đối tượng này sẽ quản lý vật lý cho đối tượng trong Game. Sau khi đã khởi tạo đối tượng trong Game ta sẽ thông qua đối tượng này để tạo nên một đối tượng bao bọc tương ứng để xử lý vật lý. Sau khi hệ thống vật lý xử lý xong ta cũng dựa vào đối tượng này để thiết lập biến đổi cho đối tượng trong Game.

Như vậy, lớp CObjectPhysicController phải có khả năng khởi tạo tất cả các đối tượng bao bọc tĩnh và động cho cả màn Game (trừ nhân vật) và truy cập vào các thuộc tính về vị trí, góc quay, trọng lượng cho các đối tượng vật lý. Ta có thể xem lớp này là một giao tiếp bao bọc để xử lý vật lý cho các đối tượng trong Game.

Lớp CCharacterPhysicController:

Lớp CCharacterPhysicController cũng có chức năng tương tự như lớp CObjectPhysicController nhưng được áp dụng cho một đối tượng đặc biệt cho phép người lập trình có thể điều khiển trực tiếp đó là nhân vật. Mỗi đối tượng thuộc lớp

CActor cũng sẽ chứa duy nhất một đối tượng thuộc lớp CCharacterPhysicController để điều khiển nhân vật. Quản lý nhân vật là một tính năng nổi bật của NovodeX được cung cấp từ phiên bản 2.2. Trước NovodeX chỉ có Navok là Engine vật lý cung cấp khả năng quản lý nhân vật. Nhờ việc xây dựng lớp CCharacterPhysicController ta sẽ sử dụng được chức năng này đơn giản hơn và hiệu quả hơn.

Với sự trợ giúp của hệ thống vật lý chúng ta sẽ tạo được những nhân vật di chuyển và tương tác tốt với môi trường. Các tính năng quan trọng điều khiển nhân vật mà NovodeX cung cấp gồm:

- ✚ Cung cấp chức năng cho phép kiểm tra và di chuyển nhân vật theo yêu cầu trong màn Game mà vẫn cho phép nhân vật tương tác với môi trường xung quanh.

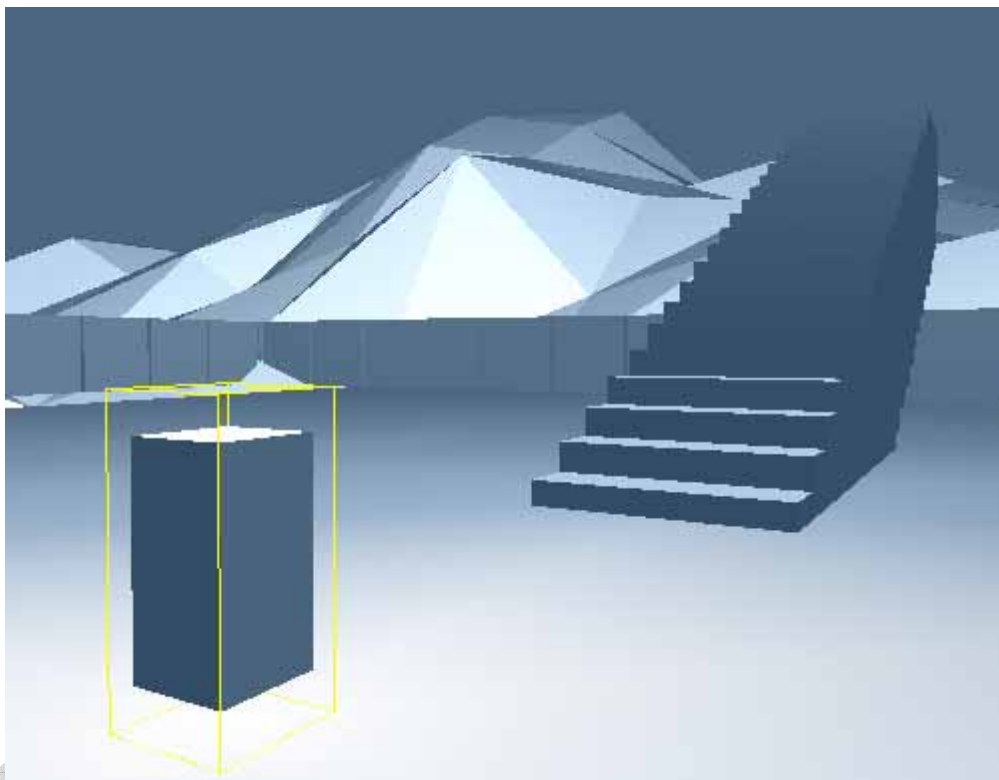
- ✚ Cho phép thực hiện các chức năng trí tuệ nhân tạo (AI) đơn giản cho nhân vật.

- ✚ Di chuyển nhân vật trượt trên nền, dọc theo các bức tường hoặc vật cản, và đặc biệt cho phép nhân vật leo lên những bậc thang với cao của các bậc thang có thể thiết lập trước.

- ✚ Thông báo và cho phép xử lý tương tác của đối tượng với môi trường như lực tác động lên đối tượng như thế nào khi có va chạm xảy ra thông qua hàm xử lý sự kiện khi có va chạm.

- ✚ Kết hợp lớp tia chiếu để nhân vật có thể nhắm bắn đạn.

Để sử dụng chức năng quản lý nhân vật trước hết ta phải khởi tạo các khối điều khiển của NovodeX bao quanh các nhân vật ngay sau khi xác định vị trí và kích thước ban đầu của nhân vật. Hình sau minh họa các bao bọc nhân vật bằng khối hình hộp:



Hình 6-3 Điều khiển nhân vật với NovodeX

Như hình vẽ, ta sẽ dùng một hình hộp để bao quanh đối tượng. Khi cần thực hiện di chuyển nhân vật ta sẽ thực hiện di chuyển hình hộp bao quanh bằng hàm `move` do NovodeX cung cấp. Thực chất khi ta gọi hàm `move` để di chuyển hình hộp điều khiển thì NovodeX sẽ thực hiện kiểm tra xem nhân vật có thể di chuyển đến điểm mong muốn không bằng cách sử dụng một hình hộp thử và di chuyển hình hộp thử này đến vị trí cần đến. Sau quá trình kiểm tra như vậy, NovodeX sẽ xác định được vị trí cũng như vận tốc di chuyển tiếp theo của khối hộp và sẽ thực hiện di chuyển thật sự.

Sau khi điều khiển hình hộp điều khiển di chuyển ta sẽ lấy về vị trí và góc quay mới để thiết lập cho đối tượng.

6.5. Tóm tắt

Engine vật lý ngày càng là một thành phần quan trọng trong xây dựng Game. Vật lý đã tạo ra bộ mặt mới cho Game, làm cho Game thêm sống động và chân thực. Với vai trò quan trọng của vật lý như vậy, chúng tôi đã tìm hiểu và sử dụng Engine vật lý NovodeX trong Game của mình. Nhờ có hệ thống vật lý mà quá trình xây dựng Game giảm bớt nhiều công sức và tăng tính hiệu quả bởi vì các việc kiểm tra va chạm đã được NovodeX xây dựng và tối ưu trên phần cứng.

Ngoài việc quản lý vật lý cho các đối tượng tĩnh và động trong Game, hệ thống vật lý xây dựng dựa trên NovodeX 2.2 cho phép điều khiển nhân vật rất thuận tiện và dựa vào đó xây dựng được các AI đơn giản cho nhân vật. Với việc kết hợp với tia chiếu do NovodeX cung cấp, chúng ta điều khiển tốt nhân vật trong môi trường tác với các đối tượng khác trong môi trường và có thể nhắm và bắn đạn chính xác.

Chương 7 Giới thiệu Game demo Dead Rising

- ◇ [Giới thiệu Game demo Dead Rising](#)
- ◇ [Nội dung cốt truyện](#)
- ◇ [Các thành phần chính cần sử dụng](#)
- ◇ [Hệ thống các tập tin định nghĩa](#)
- ◇ [Tóm tắt](#)

7.1. Giới thiệu Game demo Dead Rising

Game demo Dead Rising là ứng dụng Game được phát triển với mục đích minh họa cho các tính năng của Nwfc Engine ngoài ra Game này còn tích một số tính năng mới giúp tăng cường chất lượng đồ họa và Gameplay.

7.2. Nội dung cốt truyện

...Năm 2175, khoa học kỹ thuật trên trái đất vô cùng phát triển đặc biệt là ngành khoa học vũ trụ. Trái đất hiện tại đã bị con người tàn phá nặng nề và mức độ ô nhiễm vô cùng khủng khiếp. Nhận biết rằng trái đất sẽ bị hủy diệt trong thời gian tới, con người đã tiến hành nghiên cứu một dự án vô cùng khổng lồ: đưa toàn bộ loài người lên sao Hỏa. Dự án này mau chóng đã được hội đồng Liên minh các hành tinh (tiền thân là tổ chức Liên Hiệp Quốc) chấp nhận. Các kế hoạch nhanh chóng được triển khai, đầu tiên con người cho xây dựng một căn cứ tiền trạm tại sao Hỏa để nghiên cứu tình hình môi trường và không khí tại đây, mục tiêu chính của căn cứ này là phải nghiên cứu các cấu trúc địa tầng của sao Hỏa để tìm ra sông băng, loại vật chất duy nhất có thể sản xuất ra nước trên sao Hỏa. Trong quá trình khai thác và tìm kiếm trên các cấu trúc địa tầng hàng triệu năm tuổi của sao Hỏa con người phát hiện một số cấu trúc kiến trúc vô cùng kỳ lạ, các nhà khoa học đã xác định đây là vết tích của nền văn minh vô cùng cổ xưa, tồn tại trước khi loài người xuất hiện hàng triệu năm. Càng nghiên cứu con người đã khám phá ra nền văn minh ấy có nền khoa học kỹ thuật phát triển gấp nhiều lần trái đất bây giờ và đã có thời kỳ dân số vô cùng đông đúc. Nhưng các nhà nghiên cứu cũng phát hiện ra một điểm vô cùng kì lạ: chỉ trong một thời gian ngắn, toàn bộ cư dân của nền văn minh ấy hoàn toàn biến mất mà không để lại vết tích gì, ngay cả một bộ xương hóa thạch cũng không. Điều gì đã xảy ra với họ? đó vẫn là điều bí ẩn không có lời giải đáp chỉ biết rằng vào lúc đó họ đã phát minh được cách mở cánh cổng vào thế giới song song. Điều này đã đặt ra một loạt câu hỏi, liệu có phải họ đã di chuyển toàn bộ dân cư của mình vào thế giới song song ấy như trái đất đang làm bây giờ

hay không, các nhà khoa học vẫn mãi loay hoay tìm lời giải đáp, cho tới một ngày...

Trung tâm chỉ huy tại trái đất liên tục nhận được các tín hiệu không rõ ràng từ trạm nghiên cứu trên sao Hỏa, ít lâu sau thì người ta không thể nhận được bất kỳ thông tin phản hồi gì tại sao Hỏa nữa, thông điệp cuối cùng mà trung tâm trái đất nhận được là lời kêu cứu khẩn cấp của một người nào đó “no..o...h.el..p..m.e”. Để xác minh điều gì đã xảy ra cho căn cứ sao Hỏa, đội đặc nhiệm Bravo Team đã được thành lập. Các thành viên của đội đặc nhiệm được tuyển chọn vô cùng gắt gao từ lực lượng lính marine (lực lượng tinh nhuệ nhất của trái đất). Nhiệm vụ của Bravo Team là tiếp cận trung tâm điều khiển chính trên căn cứ sao Hỏa, thiết lập thông tin liên lạc lại với trái đất và tìm hiểu điều gì đã xảy ra tại đó. Được hỗ trợ bởi tàu chuyên chở quân sự Dropship, Bravo Team đã nhanh chóng tiếp cận trạm nghiên cứu trên sao Hỏa. Cái mà họ nhận ra đầu tiên là chỉ là không khí yên lặng đến ghê người, hầu như không thấy một chút gì chứng tỏ là sự sống đã từng hiện diện ở nơi đây. Đến được trung tâm điều khiển chính, Bravo Team cẩn trọng đi vào, bạn chỉ là thành viên trong lực lượng hỗ trợ (Backup Team) nên được giao nhiệm vụ canh giữ cổng chính của trung tâm điều khiển. Biết bao nhiêu vũ khí tối tân hạng nặng đã đi cùng với lực lượng thâm nhập (Assault Team) chỉ để lại cho bạn một khẩu machine gun cũ kỹ với một vài băng đạn. Nhiệm vụ của bạn là phải luôn duy trì liên lạc với Assault Team để kiểm soát tình hình. Điều gì đến đã phải đến, bạn nghe thấy tiếng súng nổ, tiếng đổ vỡ, tiếng la hét, rên rỉ trong bộ đàm... phút chốc mọi thứ đều im bật, im như lúc ban đầu bạn đến đây vậy. Bạn không thích chuyện này, “điều gì đã xảy ra ?” trong đầu bạn cứ luôn quanh quẩn câu hỏi ấy. Quyết tâm tìm cho ra sự thật bạn quyết định tiến vào bên trong căn cứ. Với khẩu machine gun nắm chặt trên tay bạn đưa tay ấn nút mở cổng chính...

Con ác mộng đã bắt đầu... và có thể nó sẽ không bao giờ kết thúc...

7.3. Các thành phần chính cần sử dụng

Nhiệm vụ chính của Game demo Dead Rising là dùng để minh họa các chức năng của Nwfc Engine. Nhưng nếu chỉ sử dụng Nwfc Engine thì ta chỉ có thể dựng hình 3D chứ không thể tạo thành ứng dụng Game hoàn chỉnh được. Chính vì lẽ đó ta sẽ sử dụng thêm các hệ thống khác bổ sung.

✚ Hệ thống diễn hoạt: Sử dụng hệ thống diễn hoạt đã được giới thiệu và xây dựng ở chương 5.

✚ Hệ thống vật lý: Sử dụng hệ thống vật lý đã được giới thiệu và xây dựng ở chương 6.

✚ Hệ thống hạt (Particle System), trí tuệ nhân tạo (AI): Sẽ được trình bày ở chương sau.

✚ Ngoài ra ta cần thêm các hệ thống phụ như hệ thống quản lý tập tin, hệ thống quản lý âm thanh.

Trong các hệ thống trên, hệ thống diễn hoạt và hệ thống vật lý được xây dựng mang tính độc lập và tổng quát cao nên được dùng ở nhiều ứng dụng khác nhau. Chính vì vậy chúng tôi đã trình bày như một phần riêng không liên quan đến ứng dụng cụ thể.

7.4. Hệ thống các tập tin định nghĩa

Các tập tin định nghĩa (definition files) là hệ thống tập tin được phát triển từ định dạng tập tin Parameter của Engine. Hệ thống tập tin này có vai trò rất lớn trong Game vì hệ thống này giúp lưu trữ tất cả các thuộc tính và thông số của tất cả các đối tượng trong Game, từ người chơi (player), các con quái vật (monsters) cho đến hệ thống hạt (particle system), trí tuệ nhân tạo (AI)... Việc lưu trữ các thông số trong trò chơi trong các tập tin có ý nghĩa rất lớn trong việc chỉnh sửa và thay đổi sau này mà không cần phải biên dịch lại mã nguồn của toàn bộ chương trình.

Hệ thống tập tin định nghĩa trong Game Dead Rising được phân chia nhiều thành phần tùy theo chức năng sử dụng.

7.4.1. Định nghĩa giao diện (GUI)

Tập tin định nghĩa giao diện cho phép người thiết kế đồ họa thiết kế và trình bày các đối tượng giao diện trên màn hình.

```
mgun_hud
{
    matlib          "matlibs/gui/hud_lib.nmt"
    panel ammo
    {
        material    ammo_panel
        rect        [ 0 , 0 , 256 , 64 ]
        align       rb
        color       [ 0.39 , 0.57 , 0.64 , 1.0 ]
        text_org    [ 204 , 30 ]
    }
    panel ammo_filler
    {
        material    ammo_filler
        rect        [ 0 , 0 , 128 , 64 ]
        align       rb
        color       [ 0.0 , 1.0 , 0.0 , 1.0 ]
        text_org    [ 0 , 0 ]
    }
    . . .
}
```

7.4.2. Định nghĩa hệ thống hạt (Particle System)

Tập tin định nghĩa cho hệ thống hạt lưu trữ các thuộc tính hình thành nên 1 hệ thống hạt đặc trưng. Tập tin định nghĩa giúp phân rã công việc của người thiết kế và người lập trình.

```
blood_pray          smoke
{
    material          blood_pray          "matlibs/sfx/smoke_lib.nmt"
    sprites           4
    e_type            point
    e_size            0
    e_cycletime       0.1
    e_numpars         20
    e_maxcycle        5
    max_particles     200

    c_size            8.0
    c_speed           5.0
    c_spin            0.0
    c_life            2.0
    c_dir             [ 0.0, 1.0, 0.0 ]

    c_sizevar         0.5
    c_speedvar        0.5
}
```

```
c_spinvar      0.0
c_lifevar      0.5
c_dirVar       0.5

c_start_color   [ 1.0, 1.0, 1.0, 1.0 ]
c_end_color     [ 1.0, 1.0, 1.0, 0.5 ]

// smoke unique info
scale_speed     0.5
wind_force      [ 0.0, -5.0, 0.0 ]
wind_distort    0.0
}
```

7.4.3. Định nghĩa màn chơi (Map level)

Tập tin này có nhiệm vụ định nghĩa các thực thể tĩnh (static entity) cho 1 màn chơi (cái hộp, cửa, items...) Có 3 loại thực thể có thể được định nghĩa:

✚ **entity**. Định nghĩa thực thể tĩnh

✚ **door**. Định nghĩa các cửa tự động

✚ **item**. Định nghĩa các vật dụng có thể tương tác với người chơi như bình máu, hộp đạn

```
entity      1
{
    object    tbox1
    pos       [ -183 , 66 , 79 ]
    rot       [ 0 , 120.219 , 0 ]
}
. . .

door 1      23_1
{
    object    deldoor1
    pos       [0, 15.993, -151.98] [-85.962, 15.993, -151.98]
    rot       [ 0 , -90 , 0 ]
    sensor_range 60
    open_time 1.0
    delay_time 1.0

    sound
    {
        open  "sound/door/open3.wav"
        close "sound/door/close4.wav"
    }
}
. . .

item 1      medkit
{
    object    medkit_large
}
```

```
pos          [ -76.728 , 16.0 , -255.275 ]
rot          [ 0 , 90 , 0 ]
affect_range 10.0f
sleep_time   60.0f

sound
{
    affect    "sound/voice/voc_yeah_right.wav"
}
}
. . .
```

7.4.4. Định nghĩa đối tượng và AI

Giúp định nghĩa các tính chất của các thực thể động (npcs, monster, player ..)

```
hellknight_a
{
    body          hellknight_b
    particle      gob_drop          tongue3

    rotate_speed  2.0
    jump_height   35.0

    extents       [24, 55, 24]
    skinwidth      0.1
    slopelimit     45.0
    stepoffset     10.0
    mass           500
}
. . .
```

Định nghĩa các thuộc tính AI, âm thanh và hành động của NPCs

```
hellknight
{
    actor          hellknight_a

    // animation
    idle_a         idle2
    sight_a        roar1
    move_a         walk7
    attack1_a      leftslash        100
    attack2_a      turret_attack    100
    pain_a         headpain
    death_a        pain1

    // AI attribute
    sight_range    300
    attack_range   60
    max_health     2400

    // sounds
    sound
    {
        revive     sound/hellknight/hk_chatter_01.wav
    }
}
```

sight	sound/hellknight/sight2_3.wav
attack1	sound/hellknight/chomp1.wav
attack2	sound/hellknight/chomp1.wav
pain	sound/hellknight/hk_pain_01.wav
death	sound/hellknight/die2.wav
}	
}	

7.4.5. Các định nghĩa khác

Ngoài ra Game còn sử dụng tập tin định nghĩa cho rất nhiều mục đích khác nhau như định nghĩa các vật thể dựng hình (render model), định nghĩa các thuộc tính vật lý cho vật thể, định nghĩa vũ khí, định nghĩa người chơi...

Việc triển khai hệ thống tập tin định nghĩa có ý nghĩa rất lớn trong việc mở rộng Game sau này như ta hoàn toàn có thể tạo 1 màn chơi (level) mới hay thêm vào các vật thể mới mà không phải sửa lại toàn bộ code của chương trình rất thuận lợi để phát triển các level editor sau này.

7.5. Tóm tắt

Trong chương này, chúng ta đã được giới thiệu về Game demo Dead Rising cùng các thành phần liên quan. Chúng tôi cũng trình bày về hệ thống các tập tin định nghĩa giúp quản lý mọi thông tin của Game một cách thống nhất và cho phép thay đổi, mở rộng sau này. Trong chương tiếp ta sẽ xem xét đến 2 thành phần làm nên nét đặc trưng và lôi cuốn của Game demo Dead Rising là hệ thống hạt (Particle System) và AI.

Chương 8 Hệ thống hạt (Particle System) và AI

- ◇ [Hệ thống hạt \(Particle System\)](#)
- ◇ [Trí tuệ nhân tạo \(AI\)](#)
- ◇ [Tóm tắt](#)

8.1. Hệ thống hạt (Particle System)

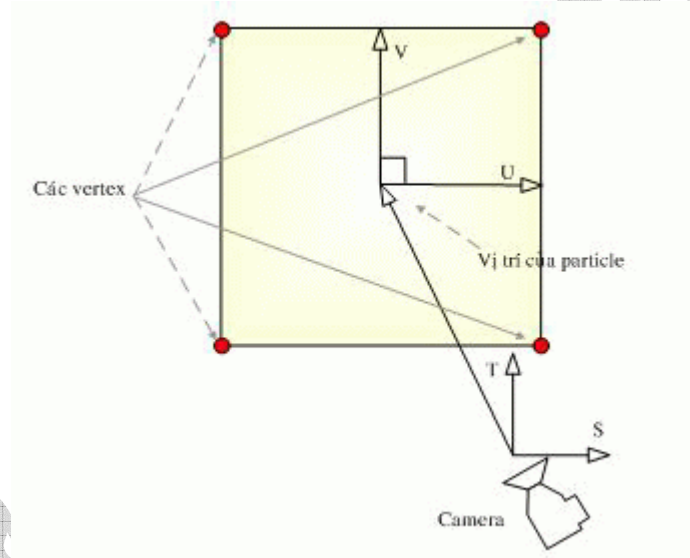
Hiện nay hệ thống hạt đã trở thành không thể thiếu trong các ứng dụng Game. Hệ thống hạt có công dụng rất to lớn trong việc thiết kế các hiệu ứng cho bối cảnh. Hệ thống hạt trong Game thường được dùng để tạo các hiệu ứng như khói lửa, cháy nổ thời tiết và các hiệu ứng phức tạp khác như sấm sét, năng lượng, ánh sáng...

Hệ thống hạt là một tập hợp các hạt (particles) có các thuộc tính và hành động hoàn toàn độc lập với nhau. Nguồn phát ra các hạt được gọi là nguồn phát (emitter), nguồn phát có chức năng cung cấp cho các hạt các thuộc tính khởi tạo ban đầu. Sau khi được khởi tạo mỗi hạt sẽ tự mình hành động độc lập theo các thuộc tính của mình cho đến khi chết đi.

Mỗi hạt trong hệ thống khi dựng hình sẽ được biểu diễn bằng 1 đa giác (thường là tứ giác). Đặc điểm của đa giác này là luôn hướng về phía camera cho dù là camera ở vị trí hay hướng nào đi chăng nữa.

Căn cứ vào mức độ phụ thuộc của mỗi hạt vào camera (khi dựng hình), hệ thống hạt trong Game chia ra làm 2 loại: hoàn toàn hướng về phía camera (smoke particle system), hướng về camera nhưng bị ràng buộc trên phương của vận tốc (spark particle system).

8.1.1. Smoke particle system



Hình 8-1 Đặc điểm của 1 particle dạng smoke

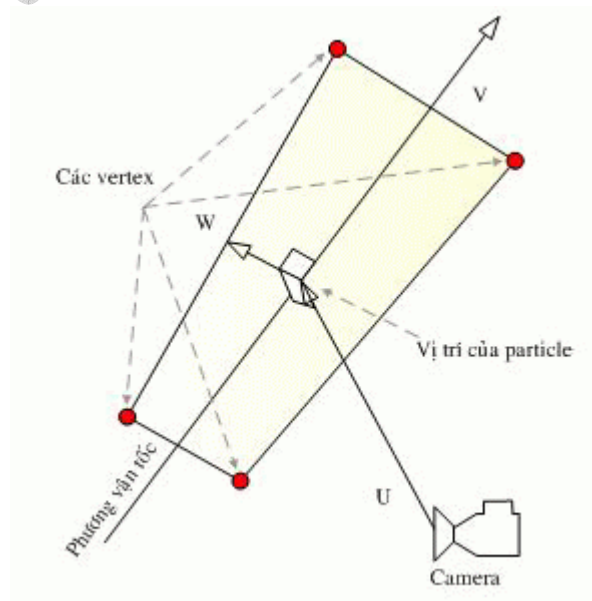
Đặc điểm chủ yếu của hệ thống này là các đa giác biểu diễn cho mỗi hạt khi dựng hình sẽ hướng hoàn toàn về hướng của camera (tức là vector pháp tuyến của đa giác song song với hướng camera). Do đó khi camera ở bất cứ đâu thì ta cũng thấy hạt là một hình vuông.

Trong smoke particle các vector V song song với T , và vector U song song với S (với S, T là hệ trục tọa độ của camera).

Hệ thống hạt này được dùng để giả lập các hiệu ứng giống khói như khói, máu, lửa, năng lượng...

8.1.2. Spark particle system

Ở hệ thống này đa giác của các hạt chỉ hướng về phía camera một phần (do các hạt bị ràng buộc trên 1 phương cố định, thường là phương của vận tốc), với đặc tính như vậy khi camera có vị trí và hướng nằm trên phương của hạt thì ta thấy hạt chỉ còn là một đường thẳng.

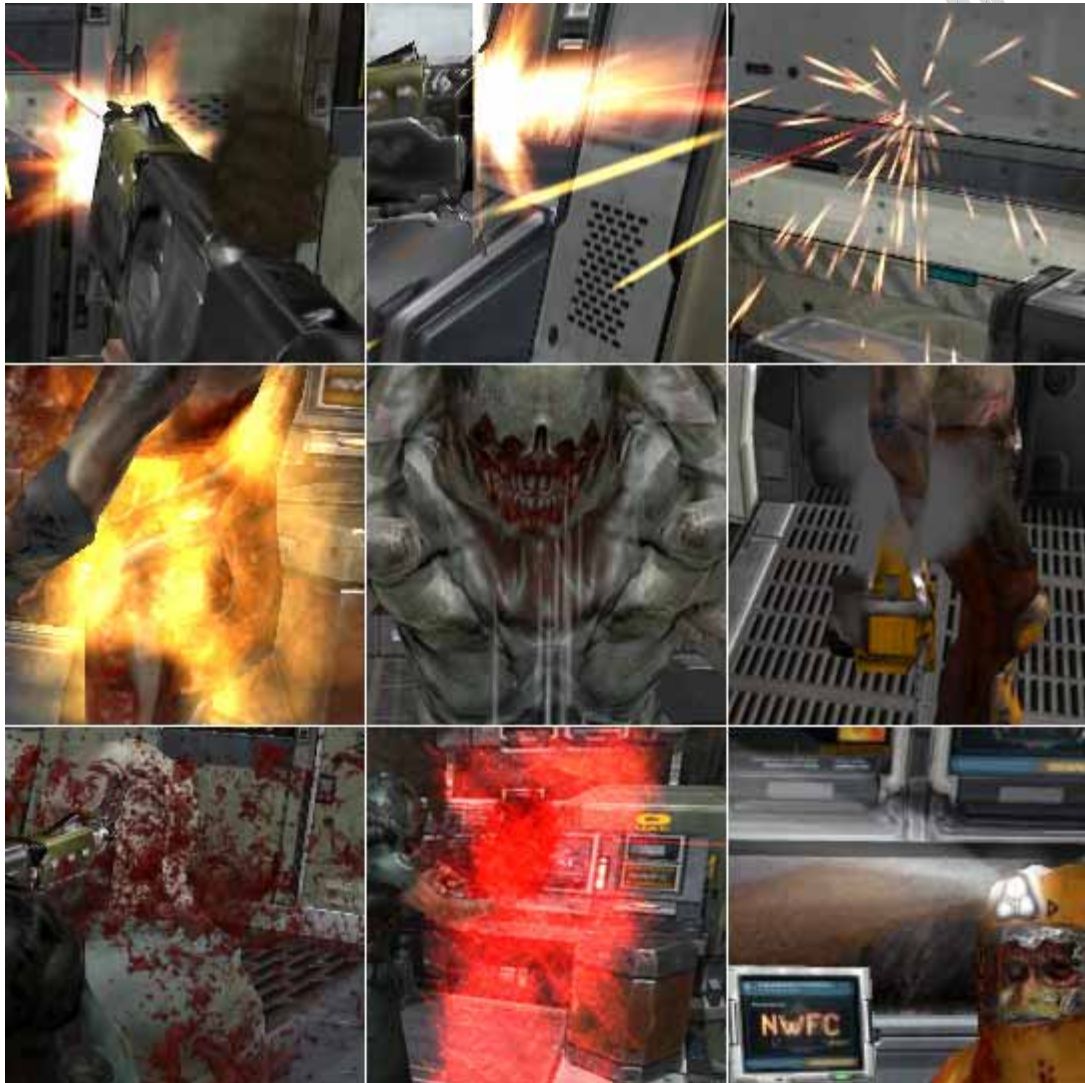


Hình 8-2 Đặc điểm của 1 particle dạng spark

Trong spark particle các vector U, V, W hợp thành hệ trục tọa độ vuông góc từng đôi một, do đó diện tích của hạt hướng về phía camera luôn là lớn nhất.

Hệ thống hạt này được dùng để giả lập các hiệu ứng giống như dạng tia lửa như tia lửa phát ra khi va chạm, tia lửa phát ra khi đạn thoát ra từ nòng súng, giọt nước, đuôi tên lửa... (hầu hết các hệ thống trên đều có tính chất là các hạt dẹp và dài trên phương của vận tốc).

8.1.3. Một số hệ thống hạt được sử dụng trong Game



Hình 8-3 Một số hệ thống hạt được sử dụng trong Game

Trong đó (từ trái qua phải, từ trên xuống dưới)

- ✚ **Hình 1.** Khói bốc lên khi bắn súng
- ✚ **Hình 2.** Tia lửa chớp khi đạn ra khỏi nòng súng
- ✚ **Hình 3.** Tia lửa xuất hiện khi đạn va chạm với tường
- ✚ **Hình 4.** Lửa bốc lên từ người quái vật
- ✚ **Hình 5.** Nước bọt chảy ra từ miệng quái vật, tạo cảm giác rùng rợn
- ✚ **Hình 6.** Khói bốc lên từ chiếc cửa máy
- ✚ **Hình 7.** Máu văng ra từ người quái vật khi bị bắn trúng
- ✚ **Hình 8.** Hiệu ứng ánh sáng khi quái vật chết hay xuất hiện
- ✚ **Hình 9.** Ánh sáng phát ra từ ngọn đèn

8.2. Trí tuệ nhân tạo (AI)

Trong Game Dead Rising có triển khai một hệ thống trí tuệ nhân tạo đơn giản để điều khiển các nhân vật phụ (NPCs – Non Playable Characters) trong Game mà chủ yếu là những con quái vật (monsters).

8.2.1. Cơ sở lý thuyết hành vi

Mỗi con quái vật trong Game là một bộ óc biết suy nghĩ và hành động theo bản năng. Bản năng của quái vật hình thành hệ thống AI cố định trong Game. Toàn bộ hệ thống AI của quái vật có thể tóm tắt như sau.

Các hành vi của quái vật đều phụ thuộc vào vị trí của nó đối với người chơi.

Mỗi quái vật đều có 2 thuộc tính quan trọng là phạm vi nhìn thấy (hay nghe thấy), và phạm vi tấn công.

✚ Phạm vi nhìn thấy. Là tầm nhìn hay bán kính tối đa mà quái vật có thể nhìn thấy người chơi. Giá trị này phụ thuộc vào độ cao và thuộc tính của mắt.

✚ Phạm vi tấn công. Là bán kính tối đa mà quái vật có thể tấn công người chơi. Giá trị này phụ thuộc vào vũ khí sử dụng và đặc tính của hành động tấn công được sử dụng (tầm xa, độ cao...)

Quá trình người chơi từ bên ngoài đi trong phạm vi nhìn thấy của quái vật sẽ kích hoạt hành loạt các hành vi của quái vật.

✚ Quái vật sẽ tìm mọi cách để xoay hướng nhìn của mình về phía người chơi, theo AI quái vật sẽ tìm cho mình góc xoay nhỏ nhất để có thể quay về phía người chơi 1 cách nhanh nhất. Thời gian thực hiện quá trình này phụ thuộc rất lớn vào tốc độ xoay của quái vật.

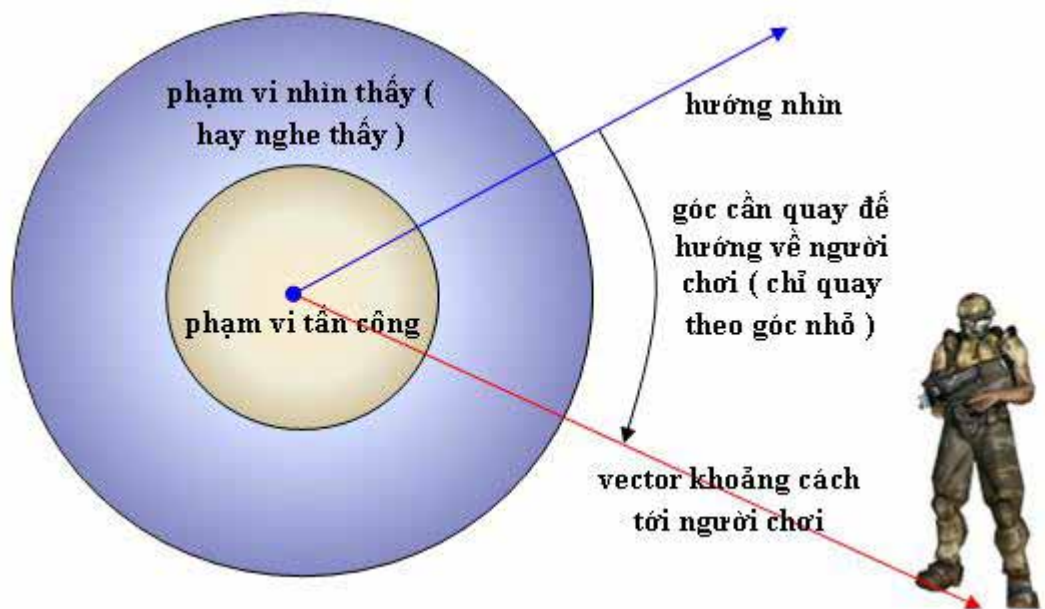
✚ Sau khi đã hướng về phía người chơi, quái vật sẽ đe dọa hay làm khiếp sợ người chơi (có thể bằng nhiều phương pháp như rống to, hay khởi động vũ khí chẳng hạn).

✚ Kế tiếp quái vật sẽ tiến về phía người chơi để tấn công. Hành động này có thể có thời gian thực hiện dài (tùy vào khoảng cách của người chơi và tốc độ của

quái vật) do đó trong quá trình này quái vật sẽ luôn tự cập nhật hướng nhìn của nó tới người chơi để đảm bảo quãng đường phải đi luôn ngắn nhất.

✚ Sau khi người chơi đã nằm trong phạm vi tấn công của quái vật, quái vật sẽ chuyển sang hành động tấn công người chơi (mỗi quái vật có thể sử dụng nhiều hành động tấn công khác nhau với mức độ sát thương khác nhau). Nếu trong quá trình tấn công mà người chơi ra khỏi phạm vi tấn công thì quái vật quay lại bước 3 (tiến về phía người chơi).

✚ Chuỗi hành vi của quái vật kết thúc khi người chơi ra khỏi phạm vi nhìn thấy của quái vật hay quái vật bị chết.



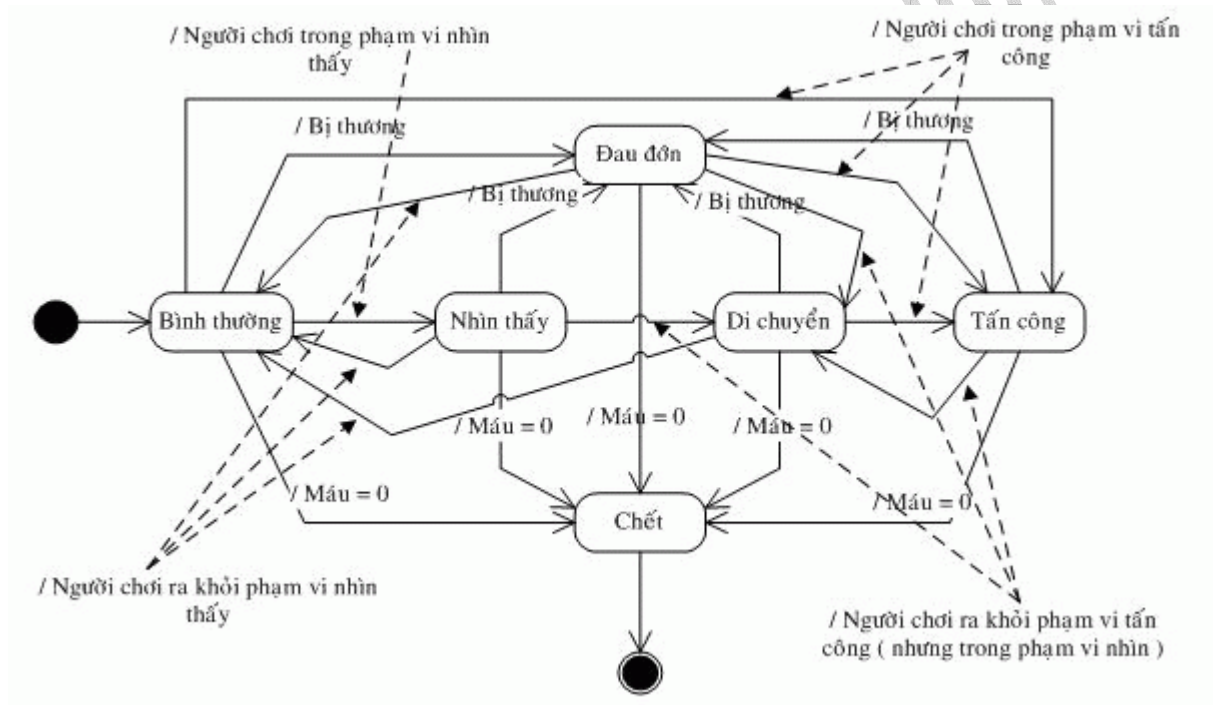
Hình 8-4 Các thuộc tính biểu diễn cho hành vi của quái vật

Trong đó:

- ✚ **hướng nhìn.** Hướng nhìn hiện thời của quái vật.
- ✚ **vector khoảng cách tới người chơi.** Biểu diễn độ sai biệt giữa vị trí hiện hành của người chơi và vị trí của quái vật.
- ✚ **góc cần quay để hướng về người chơi.** Góc biểu diễn sự sai biệt giữa vector hướng nhìn hiện hành và vector khoảng cách tới người chơi.

8.2.2. Sơ đồ trạng thái

Mỗi quái vật trong Game được biểu diễn bằng 6 trạng thái.



Hình 8-5 Sơ đồ trạng thái của quái vật

Các trạng thái

Tên trạng thái	Ý nghĩa
Bình thường (Idle)	Trạng thái bình thường, quái vật đứng yên không di chuyển
Nhìn thấy (Sight)	Nhìn thấy người chơi, quái vật thực hiện hành vi đe dọa người chơi
Di chuyển (để tấn công) (Assault)	Di chuyển hướng về phía người chơi để tấn công
Đau đớn (Pain)	Quái vật đau đớn khi bị thương và mất máu
Tấn công (Attack)	Quái vật tấn công người chơi bằng vũ khí hay tay không
Chết (Death)	Quái vật bị chết

Bảng 8-1 Các trạng thái của quái vật

Các hành động

Tên hành động	Ý nghĩa
Bị thương	Quái vật bị thương do bị người chơi bắn trúng
Máu = 0	Máu của quái vật = 0
Người chơi trong phạm vi nhìn thấy	Vị trí của người chơi nằm trong phạm vi nhìn thấy của quái vật
Người chơi trong phạm vi tấn công	Vị trí của người chơi nằm trong phạm vi tấn công của quái vật
Người chơi ra khỏi phạm vi nhìn thấy	Vị trí của người chơi nằm ngoài phạm vi nhìn thấy của quái vật
Người chơi ra khỏi phạm vi tấn công (nhưng trong phạm vi nhìn thấy)	Vị trí của người chơi nằm ngoài phạm vi tấn công nhưng nằm trong phạm vi nhìn thấy của quái vật

Bảng 8-2 Các hành động của quái vật

8.3. Tóm tắt

Trong chương này chúng tôi trình bày 2 thành phần đặc sắc mang lại những hiệu ứng trong Game Dead Rising là hệ thống hạt và AI. Bên cạnh việc dựng hình chất lượng cao của Nwfc Engine, sự hỗ trợ từ các hệ thống, hệ thống vật lý thì 2 thành phần này tạo ra những đặc trưng riêng biệt cho trò chơi. Chính 2 thành phần này sẽ tạo sức thu hút mạnh mẽ lên người chơi. Trong chương tiếp theo ta sẽ được giới thiệu và cài đặt, sử dụng Game demo Dead Rising và một số kết quả đạt được.

Chương 9 Cài đặt và hướng dẫn sử dụng

- ◇ [Môi trường phát triển ứng dụng và các công cụ](#)
- ◇ [Kết quả đạt được](#)
- ◇ [Hướng dẫn sử dụng](#)
- ◇ [Tóm tắt](#)

9.1. Môi trường phát triển ứng dụng và các công cụ

➤ **Nwfc 3D Engine được phát triển trên các môi trường và công cụ sau:**

✚ Môi trường phát triển: Microsoft Visual C++ .NET 2002, DirectX 9.0c SDK Update Summer 2004

✚ Môi trường triển khai: Microsoft Windows 2000 / XP (SP1,2)

✚ Các thư viện:

- Thư viện debug và quản lý lỗi **Debugger** (www.triplebuffer.devmaster.net)

➤ **Ứng dụng Game Dead Rising được phát triển trên các môi trường và công cụ sau:**

✚ Môi trường phát triển: Microsoft Visual C++ .NET 2002

✚ Môi trường thiết kế đồ họa 3D: 3DS Max 7.0, Maya 5.0, Deep Exploration 3.5

✚ Môi trường triển khai: Microsoft Windows 2000 / XP (SP1,2)

✚ Các thư viện:

- Engine đồ họa 3D **Nwfc** (tự phát triển)
- Engine vật lý **NovodeX SDK 2.2** (www.novodex.com)
- Thư viện debug và quản lý lỗi **Debugger** (www.triplebuffer.devmaster.net)
- Thư viện âm thanh **GAudio** (www.obrazstudio.com/gaudio/)

9.2. Kết quả đạt được

➤ **Các kết quả về đồ họa**

✚ Chất lượng đồ họa và chi tiết mà Game đạt được là rất cao do có sử dụng hiệu ứng bump bề mặt bằng normal map (xem chương 4).

✚ Rất nhiều hiệu ứng được triển khai (hệ thống hạt, ánh sáng...) làm cho Game vô cùng chân thật góp phần làm tăng tính kinh dị cho Game.

✚ Bối cảnh được dựng lên rất đồ sộ gồm bầu trời, trạm nghiên cứu (trong và ngoài), địa hình sao Hỏa...

✚ Nwfc Engine đã hoàn thành xuất sắc vai trò của mình khi giúp Game phát triển trong thời gian cực kỳ ngắn (gần 2 tháng).

➤ **Các kết quả về ứng dụng**

Game hoàn toàn có thể tương tác với người chơi do sử dụng Engine vật lý hỗ trợ.

Chú ý phát triển nhiều chi tiết nhỏ nhất trong trò chơi (như cửa tự động mở, vỏ đạn văng ra ngoài, thay đổi hình dáng khi máu giảm..) làm tăng tính hấp dẫn cho trò chơi.

Game có cấp độ tăng dần theo thời gian.

AI tuy nhỏ nhưng linh hoạt, góp phần không nhỏ vào sự hấp dẫn của Game.

9.3. Hướng dẫn sử dụng

9.3.1. Các phím điều khiển

➤ **Điều khiển toàn cục**

Nº	Phím	Tác dụng
0	Tab	Chuyển đổi camera (đi theo người chơi hay tự do)
1	Space	Vào chế độ chơi (trong màn hình intro và credit)
2	Esc	+ Hiện thị màn hình credit (trong màn hình intro) + Thoát khỏi trò chơi (trong màn hình credit)

Bảng 9-1 Các phím điều khiển toàn cục

➤ **Điều khiển nhân vật**

Nº	Phím	Tác dụng
0	W	Đi thẳng về phía trước
1	S	Đi lùi về phía sau
2	A	Xoay về hướng bên trái, có thể kết hợp với phím W để vừa đi vừa xoay
3	D	Xoay về hướng bên phải, có thể kết hợp với phím W để vừa đi vừa xoay
4	Q	Bật tắt chế độ nhắm, ở chế độ này tốc độ xoay chỉ bằng $\frac{1}{2}$ tốc độ thực tế, do đó khi muốn xoay trở nhanh phải quay lại chế độ bình thường
5	R	Nạp lại đạn
6	F	Bắn
7	Shift	Kéo dài tia lazer để nhắm (chỉ có tác dụng khi trong chế độ nhắm – phím Q), ở chế độ này tốc độ xoay chỉ bằng $\frac{1}{4}$ tốc độ thực tế
8	Space	Nhảy cao, có thể kết hợp với các phím W, S, A, D để vừa nhảy, vừa xoay vừa di chuyển

Bảng 9-2 Các phím điều khiển nhân vật

➤ Điều khiển camera (chế độ đi theo người chơi)

N ^o	Phím	Tác dụng
0	Page Up	Di chuyển camera lên cao
1	Page Down	Di chuyển camera xuống thấp
2	Home	Reset camera về vị trí ban đầu
3	RMB	Xoay hướng tự do (xung quanh người chơi)
4	Mouse Scroll	Phóng to thu nhỏ

Bảng 9-3 Các phím điều khiển camera ở chế độ đi theo người chơi

➤ Điều khiển camera (chế độ tự do)

N ^o	Phím	Tác dụng
0	Up	Di chuyển camera về phía trước
1	Down	Di chuyển camera về phía sau
2	Left	Lách camera về phía trái (hướng không đổi)
3	Right	Lách camera về phía phải (hướng không đổi)
4	Home	Reset camera về vị trí ban đầu
5	Page Up	Di chuyển camera lên cao
6	Page Down	Di chuyển camera xuống thấp
7	RMB, LMB, MMB	Xoay hướng tự do

Bảng 9-4 Các phím điều khiển camera ở chế độ tự do

9.3.2. Các chế độ chơi

Khi bắt đầu chơi người chơi sẽ vào màn hình giới thiệu (intro).



Hình 9-1 Màn hình giới thiệu

Trong màn hình giới thiệu mà nhấn **ESC** thì sẽ vào màn hình tác giả (credit)



Hình 9-2 Màn hình tác giả

Nếu đang ở trong màn hình giới thiệu hay tác giả mà nhấn **Space Bar** thì sẽ vào màn hình chơi Game.



Hình 9-3 Màn hình chơi Game

Ở phía góc dưới màn hình chơi Game có một số thông tin về trạng thái của người chơi.

✚ Góc dưới trái là thông tin về máu hiện tại của người chơi. Khi mất $\frac{1}{2}$ máu, hình ảnh bề ngoài của người chơi sẽ bị thay đổi. Khi hết máu người chơi có thể tìm bình cứu thương trong khu vực hỗ trợ (Backup Storage) để bơm lại cho đầy.

✚ Góc dưới phải là thông tin về đạn của người chơi. Bao gồm số đạn hiện thời được nạp trong súng và tổng số đạn hiện có. Khi hết đạn trong súng, người chơi phải nạp lại đạn mới được bắn tiếp. Nếu hết đạn dự trữ người chơi không thể bắn được nữa, phải lấy thùng đạn trong khu vực hỗ trợ (Backup Storage) để nạp lại.

Trong quá trình chơi, người chơi sẽ gặp nhiều loại quái vật khác nhau (mỗi lần chỉ xuất hiện 1 con mà thôi), quái vật sau sẽ mạnh hơn quái vật trước nhiều lần làm tăng độ khó cho Game theo thời gian.



Hình 9-4 Người chơi sẽ gặp nhiều quái vật trong quá trình chơi

Khi chơi người chơi hoàn toàn có thể tương tác với vật thể xung quanh, Engine vật lý được sử dụng trong Game sẽ đảm bảo quá trình va chạm này theo đúng các định luật vật lý.



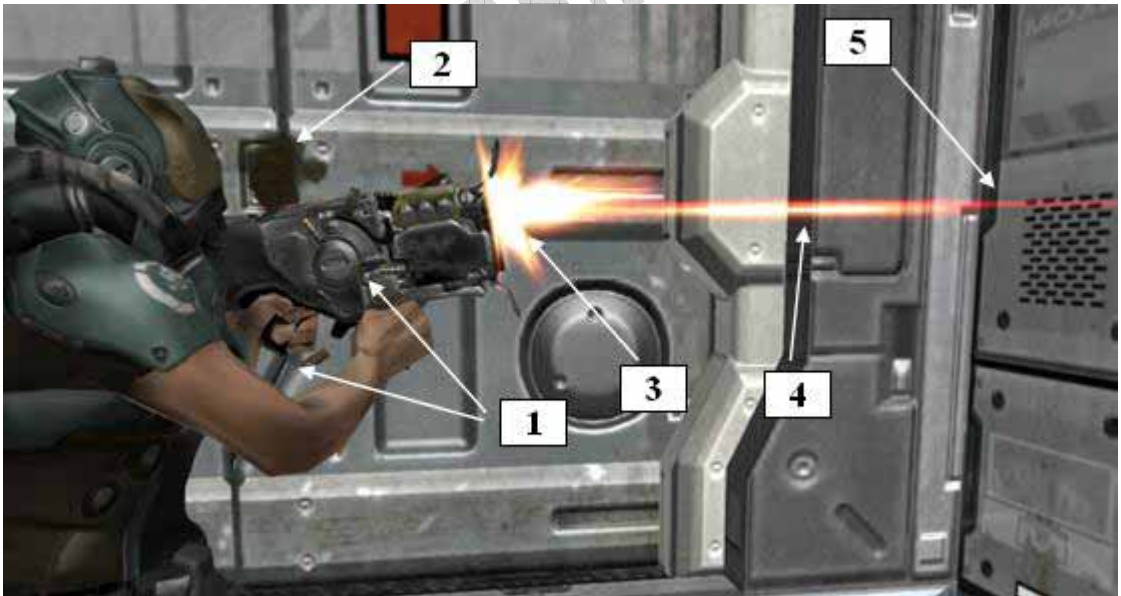
Hình 9-5 Các vật thể tương tác với nhau theo đúng các định luật vật lý

Khi người chơi đến gần các cánh cửa, cửa sẽ tự động mở



Hình 9-6 Cửa tự động mở khi người chơi đến gần

Điểm nhấn trong trò chơi chính là khẩu súng mà người chơi sử dụng. Khẩu súng được thiết kế rất nhiều chi tiết nhằm tăng cường tính thực cho Game. Khi bắn, khói từ khẩu súng sẽ bốc lên, đồng thời vỏ đạn văng ra bên ngoài (theo đúng các định luật vật lý), cùng lúc đó nòng súng sẽ chớp sáng và đạn sẽ bay ra ngoài (vì đạn bay rất nhanh nên phải để ý ta mới thấy được đường đạn). Khi đạn va chạm vào các vật thể rắn các tia lửa va chạm sẽ tỏa ra, khi đạn bắn trúng các vật thể sống thì máu sẽ phun ra.



Hình 9-7 Nhiều chi tiết được thiết kế cho khẩu súng

- [1] Vỏ đạn văng ra ngoài
- [2] Khói xịt ra ở bộ phận nòng súng
- [3] Chớp sáng ở nòng súng
- [4] Tia đạn phóng ra ở tốc độ rất lớn
- [5] Tia lazer (màu đỏ) giúp nhắm bắn chính xác



Hình 9-8 Khi bắn trúng quái vật, máu sẽ phun ra

Ngoài ra, trong Game còn sử dụng rất nhiều hiệu ứng khói lửa và ánh sáng.



Hình 9-9 Lửa bốc lên từ người quái vật



Hình 9-10 Hiệu ứng ánh sáng khi quái vật chết hay xuất hiện

Và nhiều hiệu ứng khác. Ngoài ra Game còn hỗ trợ rất nhiều tiếng động âm thanh giúp tăng cường tính thực cho Game.

9.4. Tóm tắt

Chương này trình bày về các công nghệ, thư viện được sử dụng, môi trường cài đặt và triển khai của Game demo Dead Rising. Thông qua việc hướng dẫn sử dụng chúng tôi đề cập đến những tính năng nổi bật của trò chơi thông qua một số hình ảnh được chụp từ Game. Mặc dù được phát triển trong thời gian rất ngắn nhưng Nwfc Engine và Game Dead Rising có chất lượng rất tốt với nhiều hiệu ứng phong phú.

Chương 10 Tổng kết

- ◇ [Kết luận](#)
- ◇ [Hướng phát triển và mở rộng](#)

10.1. Kết luận

Trên cơ sở những gì đã tìm hiểu và nắm bắt được từ công nghệ Shaders trên phần cứng, chúng em đã tìm cách triển khai công nghệ này để phát triển một 3D Engine mới độc lập, thích hợp cho nhiều loại ứng dụng. Để tăng cường tính năng đồ họa cho Engine, chúng em cũng đã nghiên cứu rất nhiều thuật toán Vertex Shader và Pixel Shader như Shadow Volume, per-pixel lighting... và bước đầu đã có những kết quả rất khả quan. 3D Engine được chúng em phát triển trên cơ sở tìm hiểu và phân tích nhiều khía cạnh của Engine như phân tích về mặt cấu trúc, phân tích cách phát triển thư viện độc lập theo module... do đó Engine này có thể dễ dàng được nâng cấp và có thể dùng lại cho nhiều ứng dụng đầu cuối khác nhau.

Sau khi xây dựng xong 3D Engine chúng em còn tìm hiểu và xây dựng nhiều hệ thống khác như hệ thống diễn hoạt, hệ thống vật lý, hệ thống các hiệu ứng nhằm kết hợp với 3D Engine để tạo ra các ứng dụng Game.

Về mặt chương trình, chúng em đã nghiên cứu và phát triển 1 Game demo minh họa cho khả năng của Engine. Game đã đảm bảo đúng các tính chất của một ứng dụng tuy còn chưa hoàn chỉnh lắm. Bằng cách phát triển một hệ thống tập tin riêng cho Game, các thành phần trong Game giờ đây đã hoàn toàn độc lập với mã nguồn mang lại cho Game tính khả chuyển rất cao nhằm giảm bớt gánh nặng cho người lập trình. Chúng em cũng đã tích hợp trong Game nhiều thư viện như vật lý, âm thanh... nhằm tăng cường tính hấp dẫn cho Game, góp phần tăng cường chất lượng cho sản phẩm.

Trên cơ sở những gì đã đạt được, chúng em hi vọng trong thời gian tới sẽ phát triển Engine thêm nhiều chức năng mới đồng thời cải tiến các chức năng hiện có để tăng cường khả năng triển khai ứng dụng của Engine sau này.

10.2. Hướng phát triển và mở rộng

➤ Hướng phát triển và mở rộng của Nwfc Engine

Các chức năng hiện tại của Nwfc Engine chỉ mới dừng lại ở mức đủ dùng và chưa được hoàn thiện. Trong tương lai, Nwfc Engine có thể được phát triển để hỗ trợ các tính năng đồ họa cao cấp như High Dynamic Range Lighting (HDR), Render Target Texture, Light mapping, Soft shadow... Thêm vào đó cơ chế tích hợp Vertex Shader và Pixel Shader trong Engine chưa được linh hoạt (do không thể bổ sung các hằng giá trị mới từ bên ngoài mà phải phụ thuộc vào các hằng giá trị do Engine cung cấp), do đó cơ chế này cũng cần được cải tiến và nâng cấp. Bên cạnh đó Engine còn có thể cải tiến về mặt tốc độ như tối ưu hóa các thuật toán giúp tăng tốc độ dựng hình cũng như bổ sung thêm nhiều thuật toán dựng hình tối ưu khác như Occlusion Geometry, Culling...

➤ Hướng phát triển và mở rộng của Game Dead Rising

Game Dead Rising hiện tại chỉ mang mục đích chính là demo cho khả năng phát triển ứng dụng trên nền của Nwfc Engine do đó Game sẽ thiếu rất nhiều chi tiết của 1 ứng dụng hoàn chỉnh.

Hướng mở rộng của Game trong tương lai là phát triển Game thành ứng dụng độc lập đồng thời cải tiến các thuật toán Vertex Shader cũng như Pixel Shader để Game có thể chạy được trên nhiều cấu hình máy tính khác nhau hơn.

PHỤ LỤC

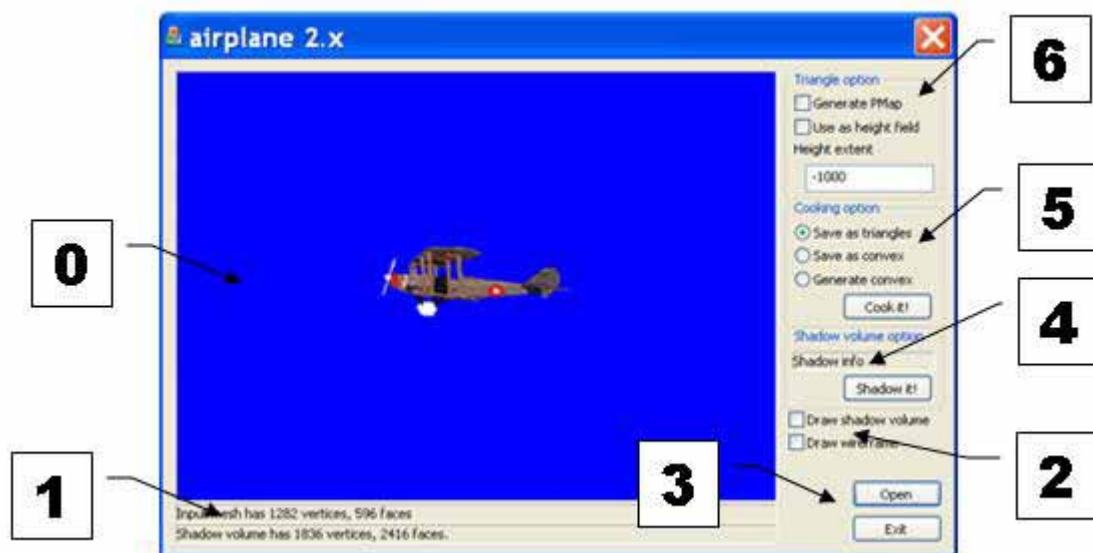
A. Hướng dẫn sử dụng chương trình MeshTools (đi kèm theo Game Dead Rising)

Chương trình MeshTools là chương trình đa chức năng được phát triển kèm theo Game nhằm mục đích hỗ trợ xử lý một số tác vụ trên dữ liệu mesh (chủ yếu là tập tin .X).


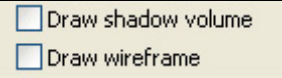
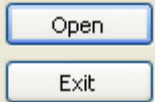
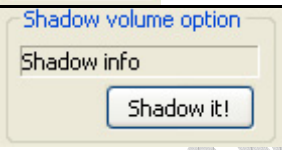
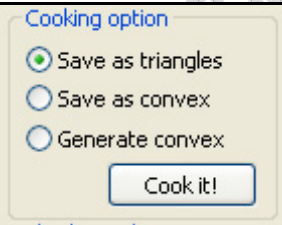
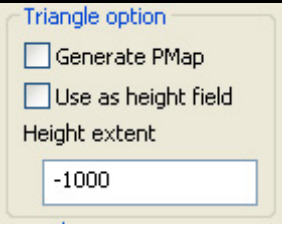
Các tính năng hỗ trợ:

- ✚ Tạo shadow volume mesh.
- ✚ Các tính năng hỗ trợ tạo mesh va chạm (collision mesh) cho Engine vật lý Novodex
 - Tạo triangle, convex mesh hay tạo convex mesh từ triangle mesh
 - Tạo pmap hay height field cho triangle mesh

Giao diện chương trình MeshTools:



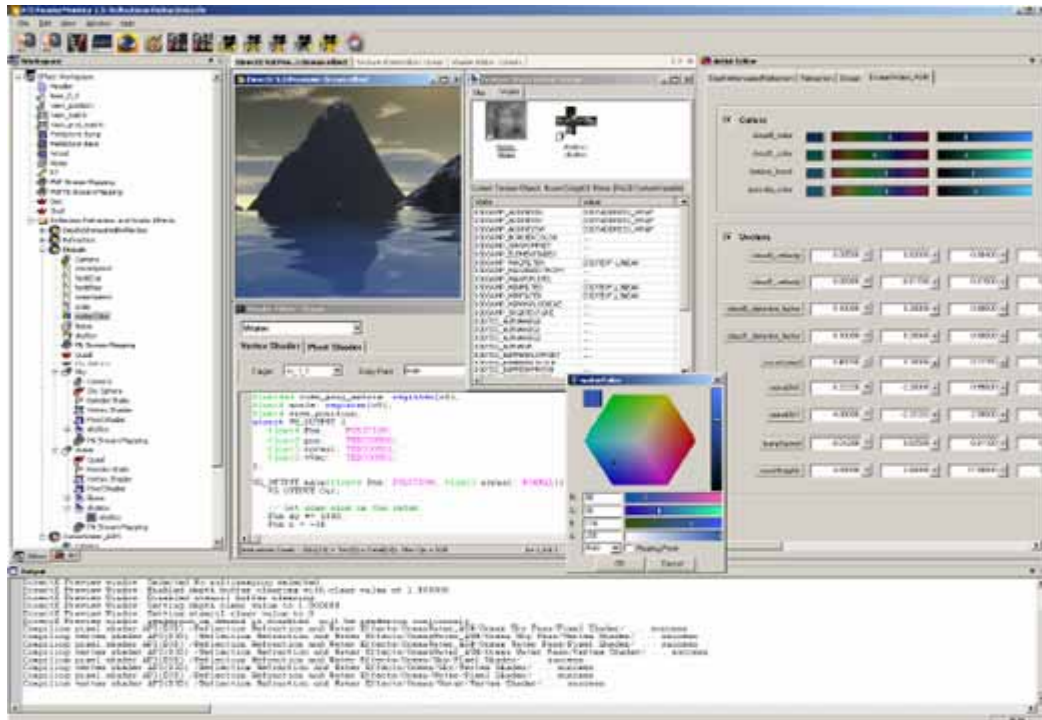
Hướng dẫn sử dụng các chức năng của chương trình MeshTools

N ^o	Hình	Chức năng
0		Khung hiển thị hình ảnh 3D
1		Thông tin xuất ra ngoài cho người dùng
2		Điều khiển hiển thị trong khung 3D + Draw shadow volume: Vẽ shadow volume mesh + Draw wireframe: Vẽ dạng khung lưới
3		+ Open: Mở tập tin 3D .X để xử lý + Exit: Thoát khỏi chương trình
4		Chức năng tạo shadow volume mesh cho thuật toán shadow volume + Shadow it!: Thực hiện lưu shadow volume mesh vào tập tin .X.
5		Chức năng tạo mesh và chạm (collision mesh) cho Engine vật lý Novodex + Save as triangles: Lưu tập tin dạng đa giác + Save as convex: Lưu tập tin dạng convex + Generate convex: Tự tạo tập tin convex từ các đa giác đầu vào. + Cook it!: Lưu vào tập tin
6		Thuộc tính bổ sung khi chọn chức năng Save as triangles + Generate PMap: Tạo thêm tập tin chứa dữ liệu PMap + Use as height field: Dùng đối tượng như dạng địa hình + Height extent: Độ sâu của địa hình (nếu dùng Use as height field)

B. Các chương trình hỗ trợ thiết kế Vertex Shader và Pixel Shader

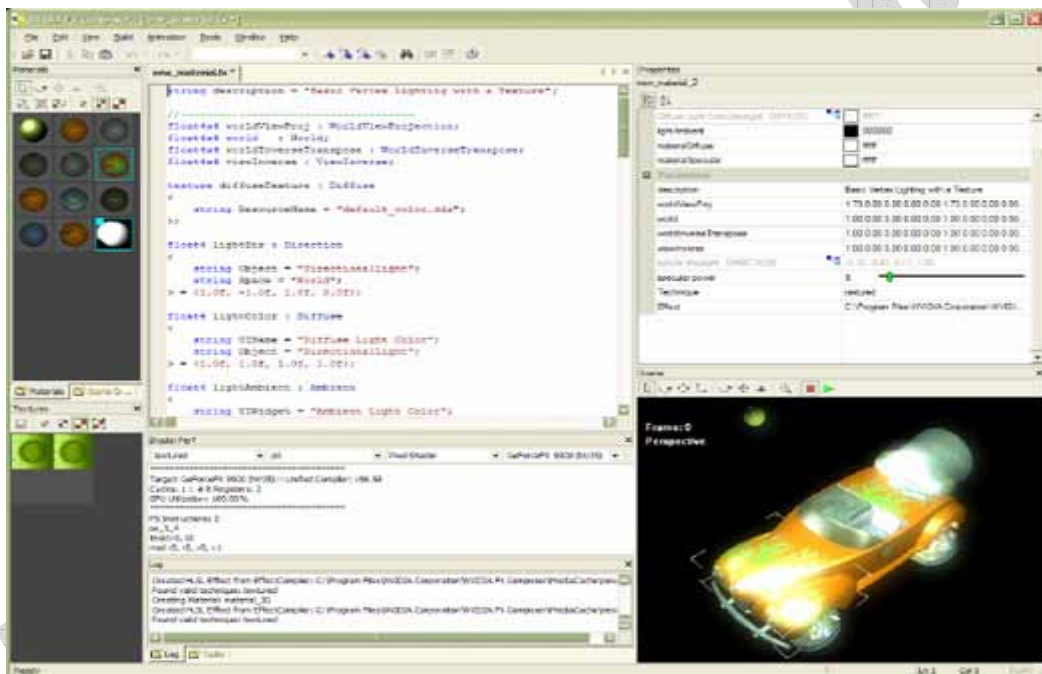
ATI RenderMonkey IDE (www.atl.com)

ATI RenderMonkey là ứng dụng IDE giúp người lập trình và người thiết kế có thể thiết kế các phương pháp dựng hình (dùng Vertex và Pixel Shader) hoàn toàn độc lập với ứng dụng. Giao diện chương trình ATI RenderMonkey IDE



NVIDIA FX Composer (www.nvidia.com)

NVIDIA FX Composer giúp người lập trình thiết kế Vertex Shader và Pixel Shader trên nền của định dạng tập tin Effect (FX file). Giao diện của NVIDIA FX Composer:



C. Engine vật lý NovodeX

Sơ lược về NovodeX:

NovodeX là Engine vật lý của công ty NVIDIA. NovodeX là một hệ thống vật lý được xây dựng thành từng gói riêng biệt nhằm tạo sự dễ dàng trong việc sử dụng và tích hợp với các Engine khác để tạo nên một Game hoàn chỉnh.

NovodeX xử lý hầu hết các vấn đề cơ bản của vật lý cổ điển như vị trí, vận tốc, gia tốc, momen, lực, chuyển động quay, năng lượng, động lượng, ma sát, độ nảy, va chạm. NovodeX cho phép chúng ta mô phỏng rất nhiều các thiết bị, cơ cấu, máy móc, mô hình khác nhau.

Quá trình phát triển của NovodeX:

- Ngày 25/8/2002 phiên bản đầu tiên 1.4 ra đời.
- Ngày 3/10/2002 phiên bản 1.41 với các chức năng quản lý bộ nhớ, kèm một số tài liệu và dự án mẫu.
- Ngày 8/10/2002 phiên bản 1.42 có sửa các lỗi về cài đặt.
- Ngày 15/10/2002 phiên bản 1.43 có thêm nhiều tài liệu hướng dẫn, chỉnh sửa một số demo.
- Ngày 30/10/2002 phiên bản 1.44 ra đời, đây là phiên bản có nhiều đổi mới như thêm kiểm tra va chạm mesh-mesh, mesh-plane, ... Và sửa các lỗi trong tài liệu.
- Ngày 2/12/2002 phiên bản 1.45 thêm các hàm trong xử lý đối tượng.
- Ngày 9/1/2003 phiên bản 1.51.
- Ngày 21/2/2003 phiên bản 1.6 sửa và thêm các chức năng trong viewer, Collision SDK, Rigid Body SDK.
- Ngày 1/6/2003 phiên bản 1.8 tiếp tục phát triển tiếp về các thành phần đang phát triển dở dang trong phiên bản 1.6.

- Ngày 7/7/2003 phiên bản 1.91 với hàng loạt chỉnh sửa và thay đổi trong Foundation SDK, công cụ hỗ trợ và thêm các demo về bánh xe răng cưa.
- Ngày 19/2/2004 phiên bản 2.02 viết lại hầu hết tài liệu kỹ thuật.
- Ngày 8/4/2004 phiên bản 2.03
- Ngày 15/5/2004 phiên bản 2.1.
- Ngày 15/7/2004 phiên bản 2.1.1.
- Ngày 8/11/2004 phiên bản 2.1.2
- Ngày 21/3/2005 phiên bản 2.2 ra đời với nhiều sự thay đổi về cấu trúc như gộp Foundation.dll vào trong physicSDK. Thêm các chức năng về điều khiển nhân vật cùng với các demo.

Đơn vị sử dụng trong NovodeX:

NovodeX không sử dụng một đơn vị đặc biệt nào trong thế giới thực. Do đó điều quan trọng là chúng ta xác định được sự chuyển đổi từ đơn vị thực vào trong ứng dụng. NovodeX SDK sử dụng các số đo không có đơn vị để đo 3 đại lượng cơ bản là khối lượng, chiều dài và thời gian do đó ta có thể định nghĩa 3 đại lượng này theo bất kì đơn vị nào ta muốn. Thông thường ta dùng sử dụng các đơn vị như kilogram cho khối lượng, met cho chiều dài, và giây cho thời gian. Các đơn vị của các đại lượng khác sẽ được xác định dựa trên đơn vị của 3 đại lượng trên ví dụ như đơn vị vận tốc sẽ là đơn vị theo chiều dài/đơn vị theo thời gian.

Cũng như bất cứ phần mềm sử dụng các số học nào, NovodeX mặc định sử dụng các số dấu chấm động đơn (single) nên chúng ta phải chú ý luôn giữ các số trong một khoảng giá trị thích hợp để NovodeX làm việc đúng đắn.

Các actor:

Trong NovodeX các actor là yếu tố chính trong một màn. Một actor có thể đóng một trong 2 vai trò là actor tĩnh hoặc actor động. Mỗi actor có thể mang nhiều hình (shape) khác nhau trong nó. Ví dụ một actor là chiếc xe có thể có nhiều hình như

đầu xe, thân xe, các bánh xe. Chúng ta cần phải đảm bảo rằng các hình trong cùng một actor không xung đột và chạm lẫn nhau.

Thông thường các actor tĩnh có nhiệm vụ phát hiện va chạm là chính do đó ta cần phải thêm các hình vào cho nó. Còn các actor tĩnh ta thì có thể dùng để đại diện cho một điểm trong lượng nào đó do đó ta có thể không cần thêm các hình vào trong nó. Và một loại actor nữa là actor động học (kinematic), đây là một loại actor đặc biệt của kiểu actor động được di chuyển trực tiếp bởi người sử dụng.

Các khớp:

Nếu không có các khớp và các hình thì actor làm được rất ít việc. Các khớp được cung cấp để nối các actor lại với nhau. Các khớp làm cho các actor kết nối với nhau cùng di chuyển theo một qui định nào đó. Trong NovodeX cung cấp cho chúng ta nhiều loại khớp các nhau để kết nối các actor, mỗi khớp sẽ được dùng để kết nối các actor theo mục đích sử dụng của ta. Các loại khớp của NovodeX cung cấp:

- Khớp cầu (spherical joint):



Đây là loại khớp đơn giản nhất. Nó gắn 2 điểm hay 2 thành phần khớp với nhau. Một ví dụ ta sử dụng khớp cầu là khớp vai của một cơ thể. Tất nhiên chúng ta có thể giới hạn lại góc quay cho khớp cầu này.

- Khớp bao ở ngoài theo 1 trục (revolute joint):



Đây là khớp mà ta chỉ có thể xoay theo một hướng duy nhất. Do đó ta cần phải xác định điểm kết nối và vector quay cho khớp này. Các điểm nằm trên chiều dài dọc theo vector hướng sẽ không được xét đến trong loại khớp nối này. Một ví dụ của khớp nối này là ta dùng như bản lề kết nối cánh cửa với tường nhà.

- Khớp trụ (cylindrical joint):



Khớp nối trụ cho phép 2 đối tượng cùng xoay với nhau theo một trục nhất định mà thôi. Và 2 đối tượng kết nối này sẽ cùng quay một góc nhau nhau.

Một ví dụ sử dụng loại trục này là cây anten của tivi hoặc máy thu âm.

- Khớp lăng trụ (prismatic joint):



Khớp này giống như khớp trụ ở chỗ nó cũng cho 2 đối tượng cùng xoay theo một trục nhất định chỉ có điểm khác biệt là nó không đảm bảo 2 góc quay của 2 vật là như nhau.

Ví dụ vì khớp trục là ta dùng trong phệt nhún làm giảm xóc cho xe.

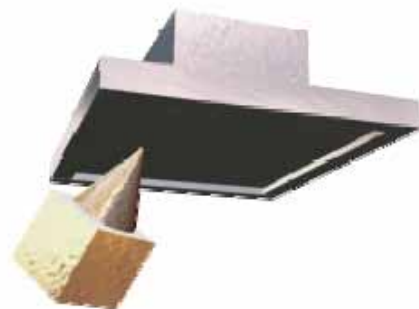
- Khớp với điểm nối trên đường:



Điểm nối trên khớp đường đảm bảo điểm kết nối của một vật chỉ di chuyển trên một đường định trước trên vật kia. Ta cần các định điểm bắt đầu và vector hướng khi khởi tạo kết nối.

Một ví dụ có thể dùng loại khớp này là cái giầy treo đồ và đồ treo.

- Khớp với điểm nối trên mặt phẳng:



Loại khớp này cho phép ta kết nối 2 vật thể với điểm tiếp xúc của vật thể này sẽ di chuyển trong một mặt phẳng của vật thể khác. Khi khởi tạo cần xác định điểm đầu tiên và vector pháp tuyến của mặt phẳng.

Ví dụ như một bút vẽ trên mặt phẳng.

- Khớp cố định (fixed joint):

Loại khớp này kết dính hai vật thể với nhau và giới hạn khoản biến thiên thay đổi giữa 2 vật thể. Khớp này rất hữu ích khi mô phỏng những cho nứt, gãy.

- Khớp khoảng cách (distance joint):

Loại khớp này sẽ cố gắng giữ giữa 2 vật luôn cách nhau một độ dài với giới giới hạn cho trước nào đó. Chúng ta có thể dùng mô phỏng phụt nhún của xe hay một sợi dây cao su.

- Khớp ròng rọc:

Khớp này có thể dùng để mô phỏng các mắc dây xích nối với nhau.

- Khớp 6 bậc tự do:

Đây là loại khớp khá tổng hợp cho phép ta có thể kết nối 2 vật theo bất kì một loại khớp nào ta có thể tưởng tượng ra. Tại mỗi bậc tự do ta đều có thể giới hạn góc quay và vận tốc cho nó.

- Khớp có thể gãy được:

Đây là loại khớp kết nối đặc biệt mà ta có thể thiết lập độ lớn lực tối đa cho nó. Khi lực tương tác vượt qua ngưỡng này thì 2 vật kết nối nhau sẽ bị gãy rời ra. Khớp này ta có thể dùng để kết nối mặt bàn với các chân bàn.

Tóm lại, NovodeX cung cấp ta rất nhiều loại khớp và các cách để thiết lập thuộc tính về độ đàn hồi, nhún nhảy cho phép ta mô phỏng rất nhiều cơ cấu phức tạp có trong thực tế.

Các loại hình bao bọc và phát hiện va chạm:

Để xử dụng NovodeX ta phải dựa trên các đối tượng cơ bản của NovodeX để bao bọc đối tượng Game và quản lý các yếu tố vật lý thông qua các đối tượng này di chuyển, phát hiện và xử lý va chạm. Các loại hình cơ bản:

- Hình cầu (lớp NxSphereShape):

Đây là loại hình học cơ bản và đơn giản nhất. Ta chỉ cần xác định tâm và bán kính cho hình cầu này.

- Khối hộp (lớp NxBoxShape):

Ta cần xác định kích thước khối hộp gồm chiều dài, chiều rộng và chiều cao. Đây là hình bao được dùng rất nhiều để bao bọc các đối tượng.

- Hình capsule (lớp NxCapsuleShape):

Hình này giống hình trụ tròn nhưng bao tròn ở 2 đầu. Chúng ta có thể dùng để bao bọc thân người (bằng cách nối nhiều hình capsule với nhau).

TÀI LIỆU THAM KHẢO

- [1] **DirectX 9.0 SDK Update Summer 2004**, Microsoft, 2004.
- [2] Jake Simpson, **Game Engine Anatomy 101**, April 12, 2002.
- [3] Bendik Stang, **Game Engines**, IMM DTU, 2003.
- [4] David H. Eberly, **3D Game Engine Design**, Morgan Kaufmann Publishers, 2003.
- [5] Wolfgang F. Engel, **Shader X² : Introductions & Tutorials with DirectX 9**, Wordware Publishing, Inc, 2004.
- [6] Wolfgang F. Engel, **ShaderX² : Shader programming tips and tricks with DirectX 9**, Wordware Publishing, Inc, 2004.
- [7] Kris Gray, **DirectX 9 Programmable Graphics Pipeline**, Microsoft Press, 2003.
- [8] William Ford, William Topp, **Data Structures with C++ Using STL Second Editor**, Prentice Hall, 2002.
- [9] Jim Adams, **Advanced Animation with DirectX**, Premier Press Game Development, 2003.
- [10] Evan Phipho, **Focus on 3D Models**, Premier Press Game Development, 2003.
- [11] Jim Adams, **Programming Role Playing Game With DirextX**, Premier Press Game Development, 2002.
- [12] Maco Monster, **Doom3 Models**, September, 2004.
- [13] Website: <http://www.obrazstudio.com/gaudio/>.
- [14] Website: <http://www.novodex.com>.
- [15] Website: <http://sourceforge.net/>.
- [16] Website: <http://www.Gamedev.net>.
- [17] Website: www.flipcode.org.
- [18] Website: www.doom3world.org.
- [19] Website: www.ati.com.
- [20] Website: www.nvidia.com