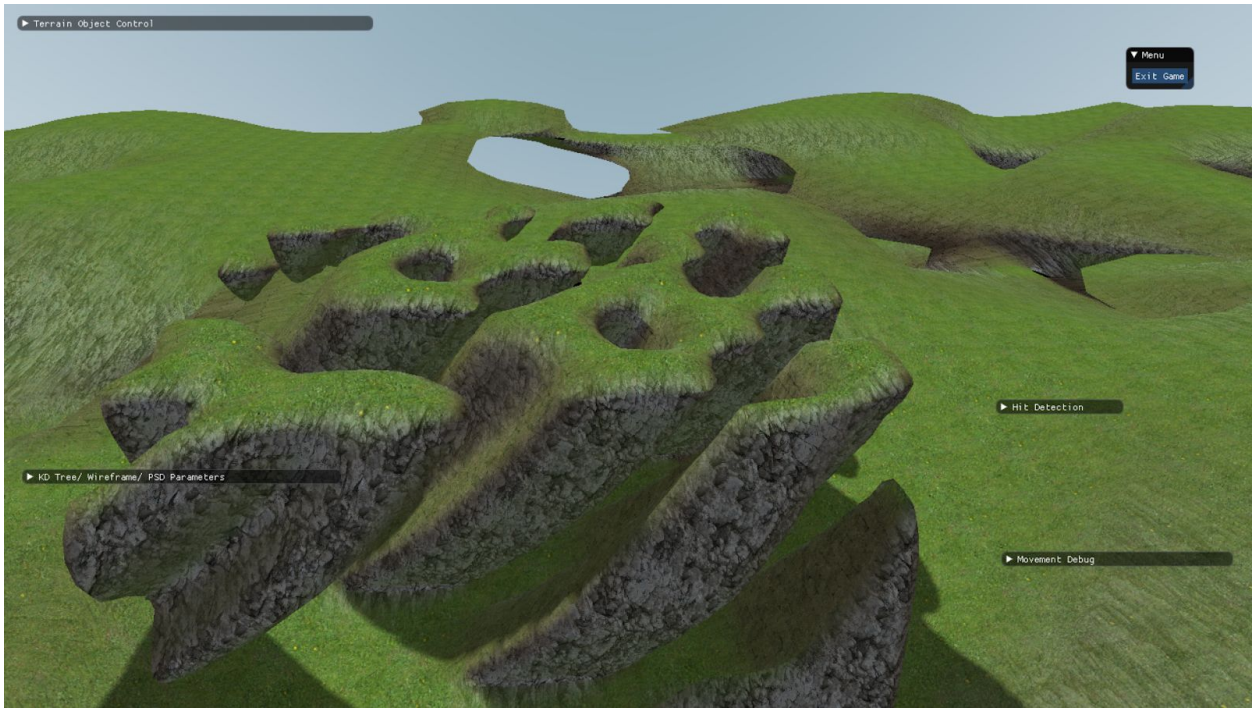
Terrain Generation with Marching Cubes and KD trees for Collision Detection



Rohan Menon

1901120@uad.ac.uk,

CMP505 - Advanced Procedural Methods,

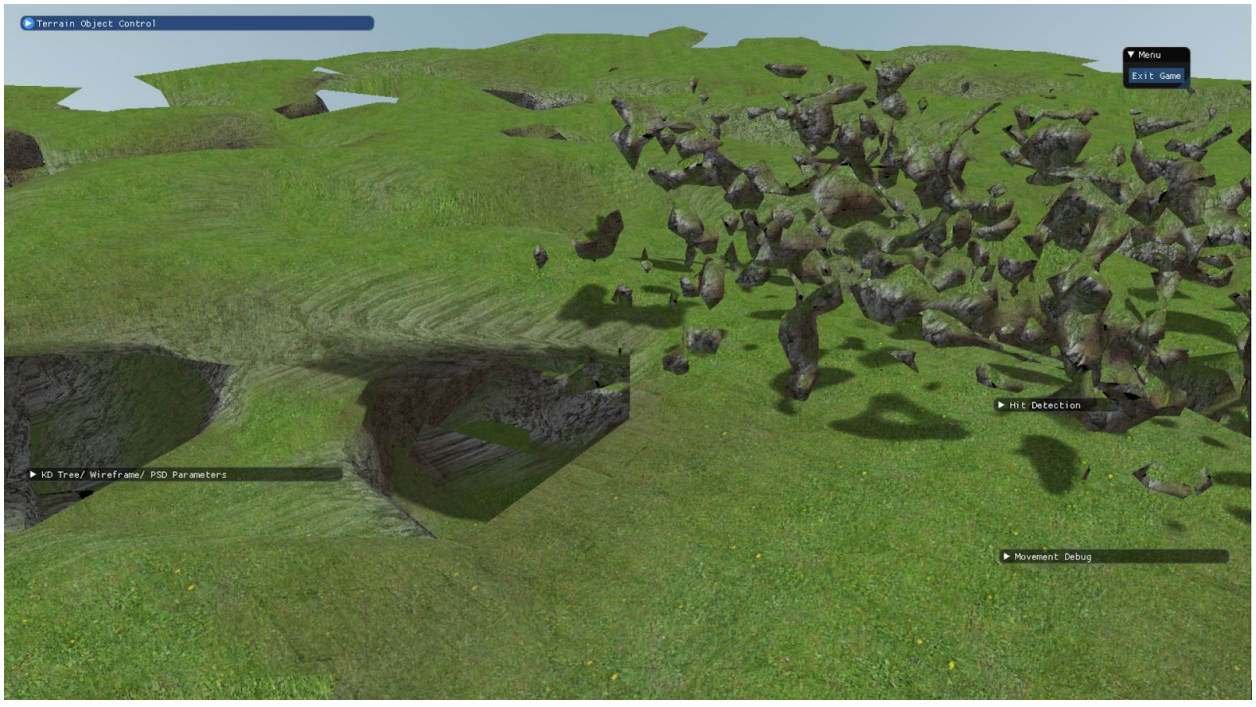
University of Abertay Dundee

INTRODUCTION

Marching Cubes a.k.a polygonising a scalar field is a tool used to generate 3D structures from a 3 dimensional grid of cube structures with different values. The intention of the project was to generate a 3D generated terrain map with an updatable object that can show different variations of terrain objects. This data was then processed in a KD Trees(K dimensional trees) for easing computation for raycasts being done whenever any of the movement controls are pressed. The terrain can be modified under the “Terrain Object Control” with different kinds of terrain spaces that can be generated. All the terrain is navigable, with restrictions on moving left, right, up and down based on 4 raycasts.

Github Repo: https://github.com/RohanMenon92/AdvancedProceduralMethods

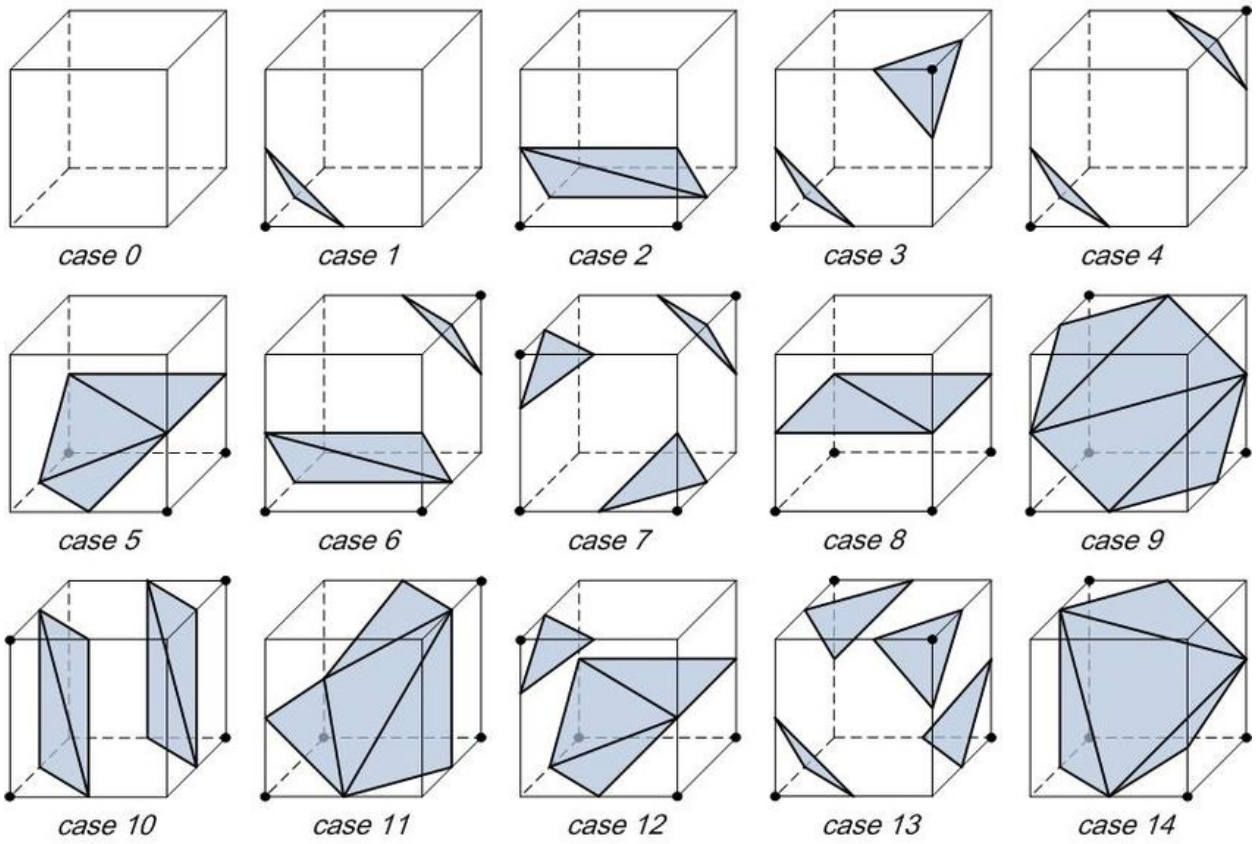
Video Link: https://youtu.be/3PMkE95prCM



MARCHING CUBES

The fundamental problem is to form a facet approximation to an isosurface through a scalar field sampled on a rectangular 3D grid. Given one grid cell defined by its vertices and scalar values at

each vertex, it is necessary to create planar facets that best represent the isosurface through that grid cell. The isosurface may not pass through the grid cell, it may cut off any one of the vertices, or it may pass through in any one of a number of more complicated ways. Each possibility will be characterised by the number of vertices that have values above or below the isosurface. If one vertex is above the isosurface say and an adjacent vertex is below the isosurface then we know the isosurface cuts the edge between these two vertices. The position that it cuts the edge will be linearly interpolated, the ratio of the length between the two vertices will be the same as the ratio of the isosurface value to the values at the vertices of the grid cell.



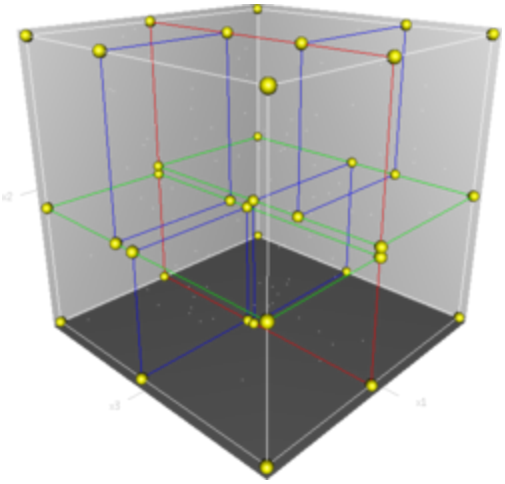
This is done through a series of lookup tables that show what vertices should be drawn for the generation of the different orientations for each scalar field value. The intersection points for the isosurface are now calculated by linear interpolation.

The last part of the algorithm involves forming the correct facets from the positions that the isosurface intersects the edges of the grid cell. Again a table (triTable) is used which this time uses the same cubeindex but allows the vertex sequence to be looked up for as many triangular facets are necessary to represent the isosurface within the grid cell. There at most 5 triangular facets necessary.

The marching cubes algorithm is ideal to be run as a shader as it allows for parallel generation of the terrain, this provides a significant boost to rendering each cube serially on a CPU. This can also be used for various other effects using shaders, particularly to render cloud geometry or randomized noise and particle effects.

KD TREE

A K-D Tree(also called a K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space. It can be visualized as a Binary Search Tree with multiple nested cubes in 3D space.



This is particularly useful if for example, we need to check whether a ray is intersecting with a particular vertex, but we do not want to check too many vertices for a ray collision and would instead like to focus on only one group that should be checked. This is also known as collision detection.

This is the exact case it is used for in the application, with a ray being moved into 4 different directions around the camera and restricting its movement to go forward, left, backward or right, based on a KDTree that is updated whenever the terrain is being generated.

The KTree can be rendered by selecting the “Render KTree” option. One sector of it is not rendered to prevent too much noise.

TRI PLANAR DISPLACEMENT

The textures being rendered on each 3D structure are a combination of 3 different sets of Rock/Grass textures each with its own basecolor and heightmap. The textures are blended based on the surface normals of the generated structure and a pixel shader is used to blend them together smoothly.

The textures themselves between the normals are also blended using Pixel Shader Displacement mapping with another pixel shader. This allows the textures to bend smoothly at sharp edges though it does create a few small texture bleeding issues.

This also allows us to smoothly tile and merge the texture when the geometry shape is being rotated or moved as well. This can be seen by modifying the movement under KDTree/wireframe. This also allows multiple generated terrains to seamlessly merge as long as they are correctly oriented to each other.

Figure 2: No PSD vs. PSD

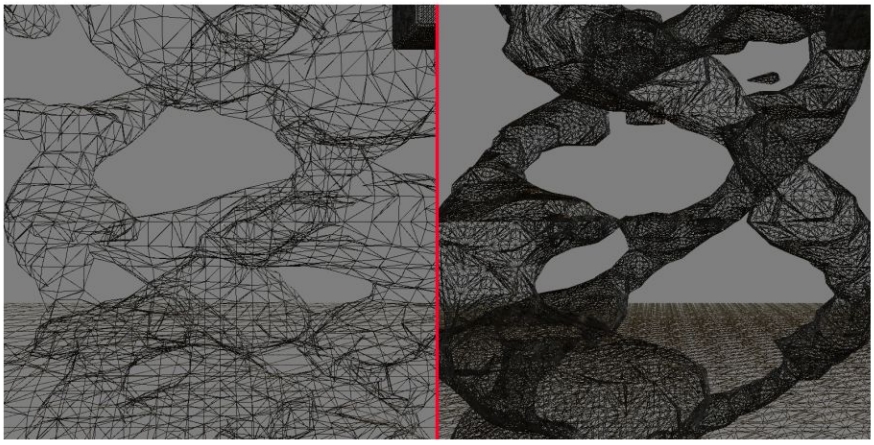
SKY DOME

A sky dome is rendered by using a Vertex and Pixel Shaders with a reversed Z buffer to render it at infinity, this allows for an understanding of orientation for the player and to tell them in which direction they are moving.

TESSELLATION

A Tessellation (or Tiling) is when we cover a surface with a pattern of flat shapes so that there are

no overlaps or gaps. Tessellation can be used to add detail to a rendered object. It is also done to allow for multiple sample sections to be defined for the generated object. It also allows for the production of smoother curves between the 18 different kinds of isosurfaces being generated from the marching cubes algorithm. The tesselation factors are currently fixed.



SOFT SHADOWS

Variance Shadow mapping is used to soften generated shadow maps and prevent jagged shadows. The shadow texture is rendered and then bound to any shaders that might need it for pixel shader data.

Variance shadow maps (VSM) replace the standard shadow map query with an analysis of the distribution of depth values. VSM employs variance and Chebyshev's inequality to determine the amount of shadowing over an arbitrary filter kernel. Since it works with the distribution and not individual occlusion queries the shadow maps themselves can be pre-filtered.

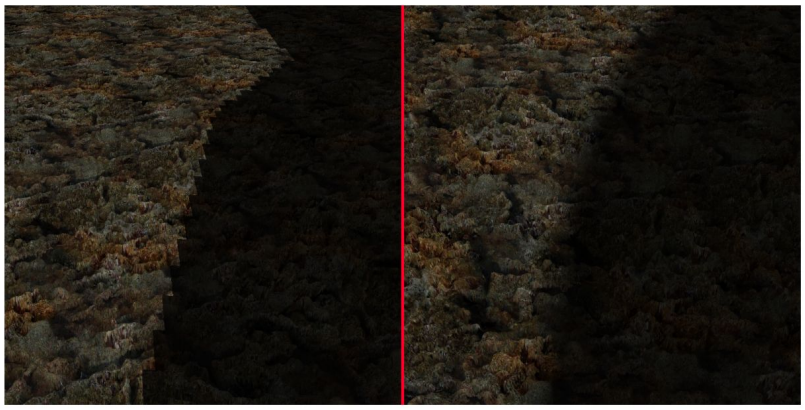


Figure 4: Hard vs Soft Shadows

CLASSES

Game.h: Runs the game and loads geometries using GeometryData.h

GeometryData.h: Uses a Marching Cube Vertex Shader, Geometry Vertex Shader for rendering 3D geometry and vertices, GeometryOutputShader to create geometry, a Tri Planar Displacement Pixel shader to create textured maps for the geometry, hullShader (Tesselation_HS) and domainShader (Tesselation_DS) for tessellation. Also references KDTree to update it.

TriangleLUT.h : Stores the lookup table for the marching cubes algorithm.

Noise.h: Class used for generating 2D and 3D simplex noise.

GeometryOutputShader.h: Used for providing the output generated by the marching cubes algorithm

ShadaowMap.h: uses VertexShader.h and PixelShader.h for generating the shadow map for the scene. Uses Depth_VS as a VertexShader and Depth_PS as a pixel shader to output colour data.

TextureClass.h: Loads a normal map and height map to texture GeometryData with.

SkydomeShader.h: Used for rendering skydome with reverse z mapping during render. Uses skydome_ps (Pixel Shader) and skydome_vs (Vertex Shader).

KDTree.h: Used to generate and render a K Dimensional tree for collision detection using raycasting.

HullShader.h & DomainShader.h: Used to run a hull Shader and domain shader for tessellation.

Camera: Stores camera position, projection, responds to movement controls

D3DClass: Rastertek example class for basic DX11 functionality

SHADERS

MarchingCube_GS.hlsl: a geometry Shader is used to generate the 3D structure from a Scalar Field created on the CPU.

Marching Cube_VS.hlsl: A Vertex Shader to render the geometry from a geometry shader into a set of vertices.

Triplanar_Displacement_PS.hlsl : A tri planar displacement mapping is used which allows slope based rendering as well as texture scaling using a pixel shader().

Geometry_VS: Used to render the geometry in GeometryData.h.

Depth_PS.hlsl & Depth_VS.hlsl: A shadow map is rendered for the scene using a Vertex shader and a pixel shader.

Tesselation_DS & Tesselation_HS: Tessellation is achieved by Domain shader and Hull shader for increased refinement in the generation.

skydome_ps.hlsl & skydome_vs.hlsl: A skydome is rendered using a pixel and vertex shader using reversed Z buffers.

CONTROLS

Key	Action
W, A, S, D	Move forward, back, left and right
Up, down, left, right	Change look direction for the camera
SPACE	Shoot to check raycast collision

PANELS

Panel	Usage
Terrain Object Control	Allows changing of the generated object type and noise variance
Hit Detection	Tells you if pressing space hit a target
Movement Debug	Tells you if you are being blocked by any terrain
KDTree/WireFrame/PSD	Allows toggling rendering of KDTree, Wireframe Tweak Pixel Shader Displacement Rotate and translate generated geometry object

TERRAIN MODIFICATION

Terrain can be modified using the “Terrain Object Control” Menu in the application. This allows modifying terrain noise scaling, the resolution of the marching cubes scalar field being constructed and also choosing the type of terrain being generated.

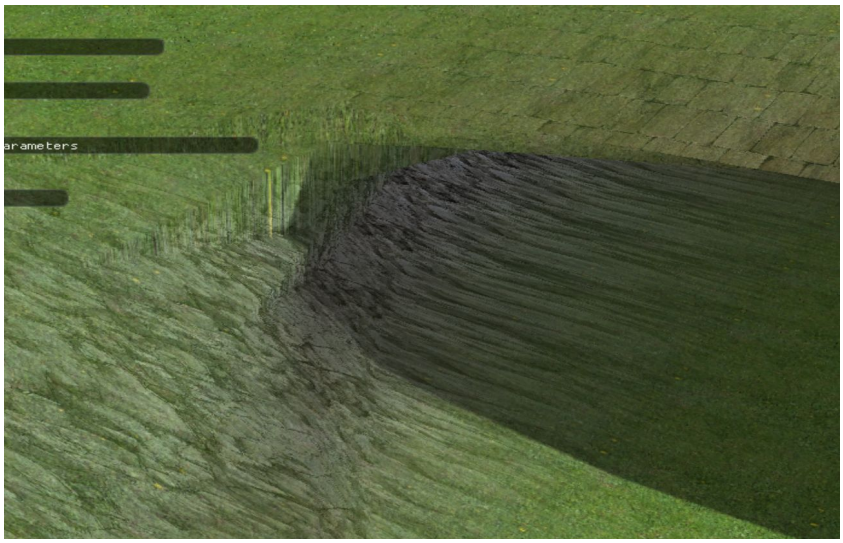
The controls are:

1. Current terrain: Which terrain is generated currently
2. Regenerate terrain: Regenerate new terrain data
3. Terrain Type: Select the terrain type you want to create. Indexed as:
    - CUBE = 0
    - NOISY CUBE = 1
    - SPHERE = 2
    - NOISY SPHERE = 3
    - 2D HEIGHT MAP = 4
    - HELIX STRUCTURE = 5
    - STRAIGHT PILLAR = 6
4. Noise Scale: How much the perlin noise function should be scaled when generating noise data. Different noise values affect the generation of the large chunk of terrain as well (Base terrain).
5. Object Resolution: XYZ controls for changing the amount of scalar fields that should be generated when performing the Marching Cubes geometry calculation. Put higher values for more intricate shapes.

ISSUES FACED

1. The shader creates a few artifacts that are created when the geometry being rendered is closer to each other. This creates a black area on render and needs to be looked into.

2. Ray cast testing with the KDTree itself is extremely expensive to do raycast tests currently. This can be refined a lot more and can also be multi threaded or perhaps written into a geometry shader later on. Another way to reduce its limitations is to reduce the frequency of checks based on frame rates (but this causes jittering issues and if the camera moves too fast, might ignore a bounding check).
3. Stretching of the texture data occurs on particularly gradual slopes, this is due to tri planar mapping and in actuality the textures being used will need to be stitched together as well(the images themselves). Currently 3 different textures are used to particularly showcase slope based texturing in the application.



FUTURE WORK

The generation algorithm itself works correctly, but since marching cubes creates volumes and not just isometric boundaries, it can be used for much more complex shaders than Tri planar texturing. It can also be used for cloud generation, calculating refraction and more complex ways of terrain stitching than just height mapping and slope mapping(density mapping as well? Sand for looser spreads and rock for more dense spreads).

The collision detection using the KDTree can be improved further, right now is quite an expensive process, this check can also be multithreaded, which should significantly reduce the amount of lag that occurs when the player reaches a high resolution KDTree branch.

The generation of the KDTree needs to be worked on as well, this is one of the most expensive processes of the generation. The actual generation of cubes and calculating the shadow map doesn't actually take that long. Perhaps outputting to another geometry shader might be an option.

The intention was to also add a blur shader to create motion blur when the camera is moving, by associating a shader or by using the BasicPostProcess.Guassian5x5 parameters but was not completed due to time constraints.

REFERENCES

Paul Bourke (1994), Polygonising a scalar field a.k.a Marching Cubes.

Available at: http://paulbourke.net/geometry/polygonise/

Jasper Flick (2017), Tri Planar Displacement Mapping(Texturing Arbitrary Surfaces)

Available at: https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/

Sebastian Lague (2019), Coding Adventure: Marching Cubes.

Available at: https://www.youtube.com/watch?v=M3iI2l0ltbE

Johannes Schauer, Andreas Nüchter (2015), Collision detection between point clouds using an efficient k-d tree implementation

Available At: https://robotik.informatik.uni-wuerzburg.de/telematics/download/aei2015.pdf

James Randall (2017), Introductory Guide to AABB Tree Collision Detection

Available at:
https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/

Matthias Moulin (2007), Rastertek coding tutorials (Particularly terrain generation)

Available at: http://www.rastertek.com/tertut14.html

Kevin Myers(2007), Variance Shadow Mapping, NVIDIA

Available at:
http://developer.download.nvidia.com/SDK/10/direct3d/Source/VarianceShadowMapping/Doc/VarianceShadowMapping.pdf