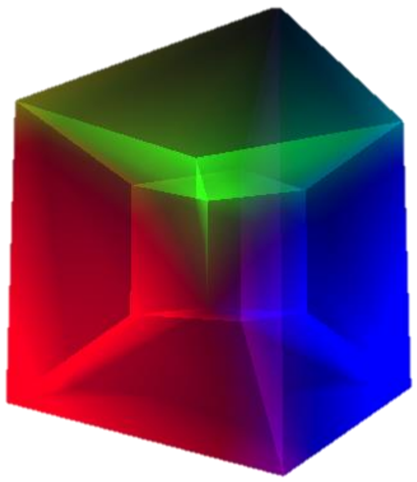


Mesh 4D Engine

A complete C# API for creating four dimensional scenes in Unity

V2.1

Contents

Introduction

Background

Mesh 4D Engine is an asset to unlock the possibility of having playable four dimensional scene in Unity. As you already know Unity is the frachise of 3D and 2D game engine which used by million game developers around the world. While being popular we took this chance to introduce and attempt to create 4D games by creating a Library/API that accept the challenge of building 4D games on top of 3D game engine.

Purpose and Motivation

The current prototype is working but in very basic form. We recommend to anyone who use this asset to have atleast average experience in building procedural objects via script for maximum usability that powers this asset.

Our primary motivation to built this asset is because, in out there countless 2D and 3D games has released worldwide, while 4D games is almost slim to none. We hope that interesting and clever 4D games will catch on our eyes after making this asset stable over time.

That being said, extending the game to another dimension is major technical problem and computationally complicated for human and expensive for computer despite major hardware upgrade and techniques in recent years. This asset is in hope to break those common hypothesis.

Our vision to this asset is that it can integrate to editor seamlessly and both programmer and designer can work with this asset natively and effectively without hurting anyone brain.

.. And of course that won’t happen if you don’t give us feedbacks!

Understanding 4-D World

The “fourth” dimension

We are living in three dimensional world, despite our eyes capture the world in 2D. Same thing happen in game engine which pushing games to function in 3D despite our screen is 2D. In mathematical abstraction we can have unlimited dimension by learning how things work in futher dimension. In this section we’ll going to explore and learn the futher dimension beyond our 3D world.

Definition of “a dimension”

A dimension is defined as a unit number of position. In 3D world you can have three dimension (we refer this as X, Y, and Z). When you move forward, your Z position is change, not X and Y (hence stay the same). When you move by fourth dimension (which usually referred as W), your XYZ number also stays the same. You can change a dimension independently without changing other dimension as well.

In another word if a human walks to fourth dimension, it is likely that he just stays the same location or suddenly disappear, depends on how you define that.

Upgrading our 3D analogy to 4D

3D objects can be refered as a point in XYZ. In 4D an object can be refered in XYZW. By default all 3D object shared the same W number unless if it change. If a person walks away to fourth dimension he would see everything in our world… flat.

A perfect explanation for understanding 4D is how we understand 3D in flat 2D world, like a paper. One of famous explanation which attempt to figure this is in a book ‘Flatland’, which tells a story where flatlanders meets with sphereland. There so many online website and videos like here, here, here, here to explain what’s in 4D world.

Deducting 4D to 3D

Everything you see in your eye is a *projection* from 3D world to 2D world *visually displayed* in your brain. It is possible to see a higher dimension with ‘limited’ information using that analogy. There are lot kinds of techniques can be used for deducting 4D worlds, though only few of them were popular enough that a lot of research came into.

Understanding 4-D (Technically)

How turning 4D to 3D world works

Actually there are two kinds of that. We call them 4D *Projection* (M4) and 4D *Cross-Section* (S4). The major difference is that M4 attempts to show all 4D shapes while S4 only show the "partial shape". In 3D analogy M4 is equivalent to how our eye or camera works, while S4 is cut the 3D shape to 2D like cutting a cube into square or a sphere to a circle.

So.. M4 or S4?

It's depends on the type of project that you are working with and how it is interact. If you would like to show all the objects and considering simplicity over challenges then use M4, otherwise S4. You also can read how they differ in next section of this documentation.

Technical challenges of 4D compared to 3D

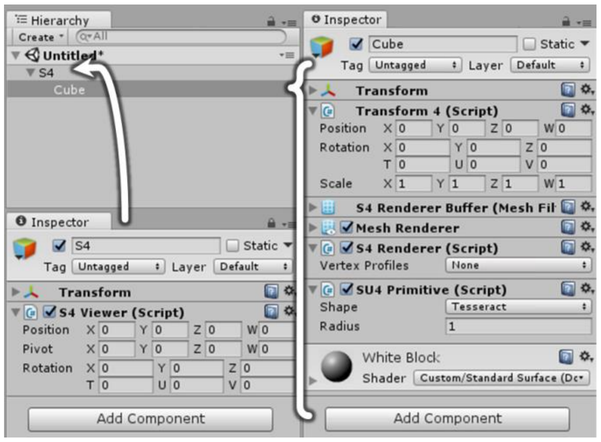
The first difference that you will know is Vector4 would be so much common operation. But much more than that you'll surprised that in 4D there are six planes of rotation you can play with (instead of three in 3D). Also in 4D you can also experiencing a gimbal lock so you have to implement another values that represents quaternion in 4D. You also need to create a value that represents Matrix5x5 for allowing translation manipulation of 4D… and the list is goes on.

For that technical reason we (internally) implements some data types (and its editor) based on existing APIs that frequently used in manipulating 3D world in the development:

Bounds4	Similar to Bounds but for 4D.
Matrix5x5	For covering 4D transformations.
Plane4	Similar to Plane but for 4D. Mainly used in S4 technique.
Quaternion4	Quaternion designed for 4D. Internally uses a pair of regular Quaternion.
Rotation4	Special struct consist of six numbers for holding euler representations
Transform4	Identical Transform component but for 4D manipulations

Workflow

Viewer-Renderer-Uploader and Transform4



S4 — Static
Tag: Untagged — Layer: Default
Transform
S4 Viewer (Script)
Position X 0 Y 0 Z 0 W 0
Pivot X 0 Y 0 Z 0 W 0
Rotation X 0 Y 0 Z 0
T 0 U 0 V 0

Cube — Static
Tag: Untagged — Layer: Default
Transform
Transform 4 (Script)
Position X 0 Y 0 Z 0 W 0
Rotation X 0 Y 0 Z 0
T 0 U 0 V 0
Scale X 1 Y 1 Z 1 W 1
S4 Renderer Buffer (Mesh Fil
Mesh Renderer
S4 Renderer (Script)
Vertex Profiles: None
SU4 Primitive (Script)
Shape: Tesseract
Radius: 1
White Block
Shader: Custom/Standard Surface (Dc

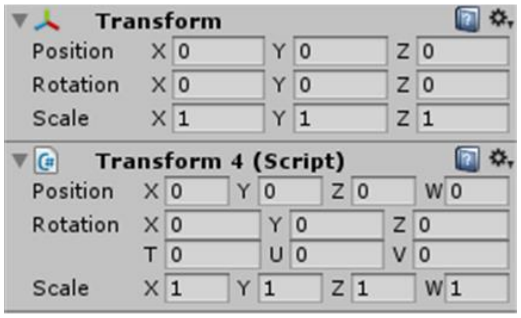
Viewer is always in the root of renderer & uploader

We use terms **Viewer-Renderer-Uploader** which has equivalent analogy with Camera-MeshRenderer-MeshFilter to make the workflow familiar. We also has **Transform4** which has similar properties and workflow with regular Transform including parenting GameObjects, etc.

In another words, The **uploader** build the shape, the **renderer** process the shape according to its **transformation** and **viewer** matrix.

The correct workflow of using these technique is there should only one viewer in the game while all 4D objects (which contains uploader+renderer) is the children of that viewer.

Transform4 vs. Regular Transform



Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Transform 4 (Script)
Position X 0 Y 0 Z 0 W 0
Rotation X 0 Y 0 Z 0
T 0 U 0 V 0
Scale X 1 Y 1 Z 1 W 1

We do not change the current transform to match 4D, instead we built new one separately. Both its position and scale is in Vector4, while the rotation is in Quaternion4 (but shown in inspector as its euler representation).

Remember that unlike regular transform, not all objects is necessarily have Transform4 unless the object has renderer inside it. Also keep a note if one Transform4 is changed, all of its children also influenced too. Just like regular transform does. And also its a good habit to just leave Transform values zero if you work on 4D scenes to keep things simple.

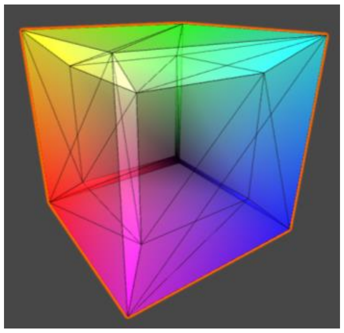
4-dimensional Rotation and its convention

It's important to know that in 4D there are no such thing about *rotation around axis* because unlike 3D, the rotation axes has different number than its vector dimensiton. The real definition of *rotation* is a rotation in given **two** dimensional plane. In mesh 4D engine, the 4D rotation signs is **X, Y, Z, T, U, V** which refers to rotation in **YZ, XZ, XY, XW, YW, ZW** plane.

The direction of X, Y, Z rotation is clockwise similar with Unity convention while T, U, V is bit different where if each of it rotated about 90° positive then each X,Y,Z axis would turn to W axis with positive sign.

Workflow: 4D Projection (M4 Technique)

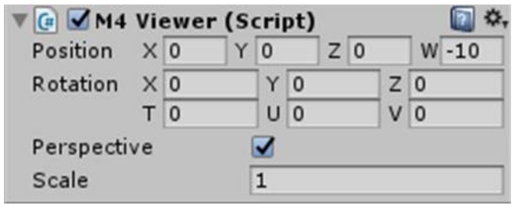
This is the most common type of projection. The function is exactly similar to your eye or camera where the farther is goes then much smaller is get. Most of the time when you are browsing the internet about 4D object you'll see the shape is displayed using this technique.



Projected Tesseract

Viewing Condition

M4 project similarly like camera. It has perspective/orthographic option. The scale option scales the projection uniformly. The important key to watch is position. The position must be negative in W because the camera must always behind the object. Make it larger (in negative) then its FOV (which automatically calculated) gets smaller, and vice-versa. If you see the projection is broken, that's because the position is too close that part of object gets behind the camera (and should be culled but it's not).



M4 Viewer (Script)

Position	X	0	Y	0	Z	0	W	-10
Rotation	X	0	Y	0	Z	0		
	T	0	U	0	V	0		
Perspective	✓							
Scale	1							

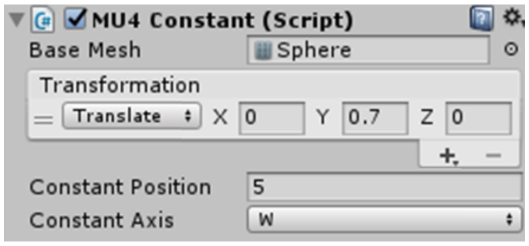
Building the shape

The plus side of this technique is that its practically easy to build a 4D shape. (Internally in Uploder) you can plug any mesh and your only job is to change those Vector3 coordinates to Vector4 in separate data, and the rest of manipulations is done by Renderer. All mesh data including UVs and color is preserved and you have an option if you like to combine submeshes or not.

Built-in Uploaders

There are two (M4) uploaders that included in this asset. Note that you can you can made your own indefinitely by reading our Scripting API in another section. Multiple uploader in the same renderer is also supported.

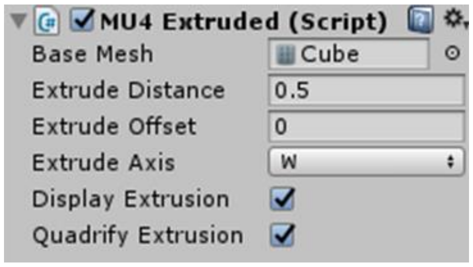
MU4 Constant Uploader



MU4 Constant (Script)
Base Mesh: Sphere
Transformation
Translate X 0 Y 0.7 Z 0
Constant Position: 5
Constant Axis: W

The simplest uploader. Plug any mesh and set the overall (constant) W value. If you want, you can also change which axes that must be constant (for example, select X then W and X swapped therefore X values will constant). Optionally you can also transform the mesh before converted into 4D shape.

MU4 Extruded Uploader

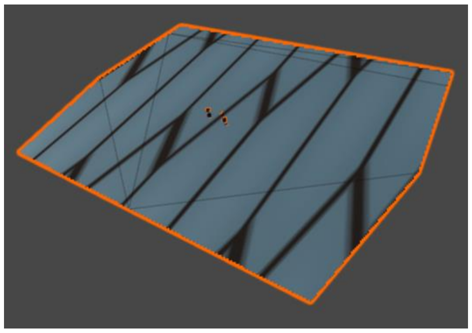


MU4 Extruded (Script)
Base Mesh: Cube
Extrude Distance: 0.5
Extrude Offset: 0
Extrude Axis: W
Display Extrusion: ✓
Quadrify Extrusion: ✓

One of basic kind of uploader. Imagine a flat shape being extruded upward, like circle becoming cylinder. This uploader does the similar job (in analogy). You can set for how long is it. Optionally by turning on Quadrify Extrusion, it can cleanup unneeded extrusion faces automagically.

Workflow: 4D Cross-Section (S4 Technique)

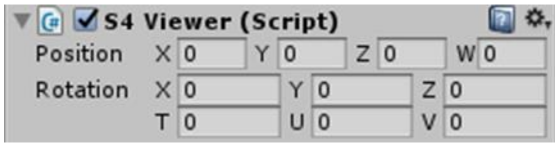
This techique is popular because it is introduced by popular novella *Flatland* by Edwin Hobbet. Based on that novel, if a sphere enters a 2D world it would seen as a cross-section of a sphere: circle, and bend and shrink particularly as it moves in a direction that 2D shapes can't touch. This analogy can be reused for higher dimension.



Bended Cross-section of Hyperplane (4D Plane)

Viewing Condition

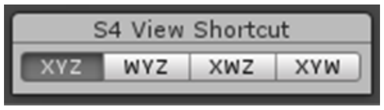
Initially, the 3D shape that generated is the cross-section of 4D world that has W = 0. Everything else just not visible. You will interact with it just normally as 3D would.



S4 Viewer (Script)

Position	X	0	Y	0	Z	0	W	0
Rotation	X	0	Y	0	Z	0		
	T	0	U	0	V	0		

Another interesting concept is when one of three 4D rotation is in right angle. In that case it's either X, Y or Z is swapped with W axis. That special case would be very useful both in game and editor because its easier for human brain to understand while walking in 4th dimension.



A Handy editor tool to swap one of XYZ with W axis for easier visualization

Building the shape

Because this technique is involving cutting a shape, building the shape requires to be defined in volume. The problem is that Meshes do defined in surface as triangles, not volume. For this reason the shape must be procedurally generated, but not by surface triangles, instead in the most simplest volume geometry: **Tetrahedron**.

Yes, like triangle, tetrahedron has 4 vertices and can be freely positioned. The great news is that if you cut a tetrahedron the result is must be either triangle or quad, which also mean that if 4D shapes that built using tetrahedrons were cutted out the result is must be 3D shape consist of triangles and quads (which forming a mesh).

The point is, because you can't make the shape from any mesh, all tetrahedron will always merged to one submesh. UVs and Colors can be included or not by individual renderer.

Built-in Uploaders

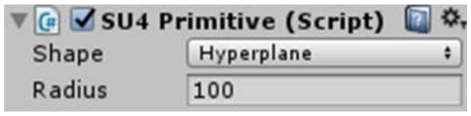
S4 technique maybe the most used 4D technique in games because its familiarity with 3D world while keeping it tidy with showing only cross-section of the so-complex 4D world compared with M4.

However, that's also means the complexity with 4D brought the pain back to its developer because building the right shape with tetrahedrons in 4D is more challenging compared when building mesh with triangles in 3D.

Only one built-in S4 Uploader at this time. Just like M4, these uploader can be extended to behave anything that comes in your mind via Scripting API.

SU4 Primitive Uploader

This uploader's main job is to create 4D primitive shapes: Hyperplane, Pentatope, Tesseract and Hexdecahedroid which its cross-section results a Plane, Tetrahedroid, Cube and Dual-pyramid. Its size can be adjusted in inspector.



SU4 Primitive (Script)

Shape: Hyperplane

Radius: 100

Scripting API

Introduction

In Mesh 4D Engine we trying to make our API much simpler and native by reusing the existing conventions in Unity3D. You would see Transform4, Quaternion4, Matrix5x5, Plane4, Bounds4.. etc.

Keep in mind to always write **`using M4DLib;`** as all APIs is bundled in that namespace.

The main part to deal with scripting in Mesh 4D Engine

There are only two reason why you need to access our asset via scripting:

1. To change and manipulate Transform4 and viewer
2. To build custom 4D shapes

Manipulate Transform4

Changing Transform4 is simple as is should be. This is the public API that available in Transform4:

```
public class Transform4 : MonoBehaviour
{
    public Rotation4 localEulerAngles;

    public Vector4 localPosition;

    public Quaternion4 localRotation;

    public Vector4 localScale;

    public Vector4 position;

    public Quaternion4 rotation;

    public Vector4 scale;

    public Matrix5x5 localMatrix;

    public Matrix5x5 localToWorldMatrix;

    public bool hasChanged;
}
```

As you see the major difference is in the value types. The major difference however, is that Rotation4 in localEulerAngles. Want to see its content?

```
public struct Rotation4
{
    public Vector3 xyz, tuv;

    public float yz, xz, xy;
    public float xw, yw, zw;

    public Matrix4x4 ToMatrix();
    public Quaternion4 ToQuaternion();
}
```

Many struct have familiar and handy operators and static functions. Most of them also have comments for extra explanation. You can explore each of them by exploring our internal code.

Manipulating Viewer

Yes! Viewer is the same, for both M4Viewer and S4Viewer. Both of them share the same Interface:

```
public interface IViewer
{
    Vector4 position { get; set; }
    Quaternion4 rotation { get; set; }
}
```

Internally changing viewer is much faster than individual transform. But when both is not changed, no update operation is performed.

Building custom 4D shapes: MU4

Custom M4 uploaders can be created by deriving from M4UploaderBase.

```
using M4DLib;

public class MU4Template : M4UploaderBase
{
    public override void UploadBuffer(M4Buffer buffer) { }
}
```

Your job is only fill that buffer. Here’s the content:

```
public class M4Buffer
{
    public Mesh mesh;

    public List<Vector4> vertices;

    public void AddMesh(Mesh m, int submesh, IEnumerable<Vector4> verts);

    public void AddMeshes(IEnumerable<Mesh> m, int submesh, IEnumerable<Vector4> verts);

    public void AddTris(int a, int b, int c);

    public int AddVert(Vector4 vert);

    public int AddVert(Vector4 vert, VertProfile profile);

    public int AddVert(int index, VertProfile profile);
    public int AddVert(int index);

    public int BeginDraw(bool useProfiles = true);

    public void DrawQuads(int a, int b, int c, int d);

    public void DrawQuads(int a, int b, int c, int d, Rect uv);

    public void DrawQuads(int a, int b, int c, int d, VertProfile A, VertProfile B,
                      VertProfile C, VertProfile D);

    public void DrawQuads(params int[] idx);

    public void DrawTris(int a, int b, int c);

    public void DrawTris(int a, int b, int c, VertProfile A, VertProfile B, VertProfile
                      C);

    public void DrawTris(params int[] idx);

    public void EndDraw(VertexProfiles profile);
}
```

Despite many functions there, the only function that you'll use mostly is **AddMesh**. That function is about upload a mesh with list of Vector4 for vertices subtitution.

However if you want more procedural approach then the rest functions is for you. The important note is always begin with **BeginDraw()** and **EndDraw()**. If you don't do it your uploads will be discarded.

If you want the example have a look on our built-in uploaders.

Building custom 4D shapes: SU4

Custom S4 uploaders can be created by deriving from S4UploaderBase.

```
using M4DLib;

public class SU4Template : S4UploaderBase
{
    public override void UploadBuffer(S4Buffer buffer) { }
}
```

Here the same job applies, but slightly different. Here's the content:

```
public class S4Buffer
{
    public List<int> trimids;

    public List<Vector4> vertices;

    public List<VertProfile> profiles;

    public void AddCube(int v0, int v1, int v2, int v3, int v4, int v5, int v6, int v7);
    public void AddTrimid(int v0, int v1, int v2, int v3);

    public void AddTrimid(Vector4 a, Vector4 b, Vector4 c, Vector4 d);

    public int AddVert(Vector4 vert);

    public int AddVert(Vector4 vert, Vector4 uv);

    public int AddVert(Vector4 vert, Color color, Vector4 uv);

    public int AddVert(Vector4 vert, Color color, Vector4 uv, Vector4 uv2, Vector4 uv3);

    public int AddVert(Vector4 vert, VertProfile profile);

    public void Clear();

    public void Expand(int predictedVert, int predictedTris);
}
```

See the difference? Yes, fully procedural. Here in S4 we use trimids (aka. Triangle pyramids/tetrahedron) and one trimid consist of four vertices (not three).

We don’t explain individual functions here because its already commented. So better explore the API now if you curious!

Legacy Components

Those stuff mentioned previously is all new way to develop 4D scenes.. since 2.x!

So what happened to existing codes in 1.x?

In 1.x we develop components that “simulate” 4D object individually. And that “simulation” is fake! It doesn’t show the right “projection” of four dimensional objects.

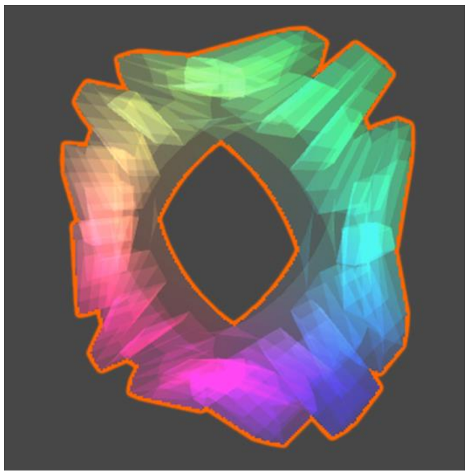
No matter what happened, it had exist for two years so it doesn’t feel right to just “abandon” it, because someone *who care about it’s artistic value* might still find the legacy components is useful.

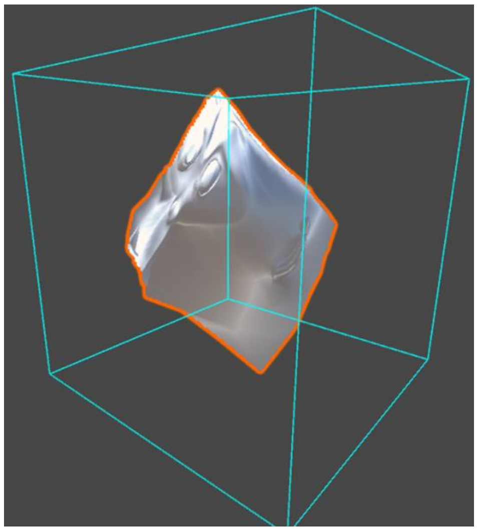
We decided to keep improve and cleanup the legacy code but won’t introduce anything new. All legacy script is bundled in **M4DLib.Legacy.**

The documentation for legacy components are still online. But if you curious…

Mesh 4D Engine

This component simulate any mesh and give extra extrusion faces similar like MU4ExtrudedUploader. In past devs must generate extrusion faces manually but now we decided to make it automatic despite being *legacy*.





Mesh 4D Morpher

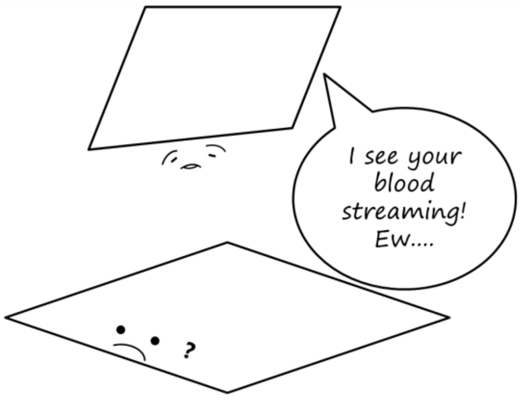
This component simulate two different mesh and interpolate them using very unique algorithm. By using that algorithm, devs can change between two different mesh in unique and bizarre way.

FAQ/Troubleshooting

Why all material must be double-sided?

Because it's expected!

Imagine there's two paper (A & B) on table in front of each other. A sees only B's skin (B's paper edges), and it's impossible to see B's internal body. Now lift paper A above paper B upside down. Now that's happening.



That's why we include a two-sided standard shader in this asset.

What about performance?

We care about it especially in very critical part of code.

For additional note, processing upload is only happens once in the game, unless somewhere in your code calls *S/M4Renderer.SetDiry(true)*. So it's okay to leave it unoptimized.

Also changing transform cost some computing time, but not that much.

If viewer and any transformation isn't change in a frame. This asset doesn't steal any computing time.

The rough average performance limit is about 100 cubes at once in each frame. Note that's "rough", anything is possible.

Why 5.6?

In 5.5.2 Unity introduces Mesh.GetXXX which can speedily load mesh data directly to List.

... and to avoid problems.

Release Notes

Jun 30, 2017 – V2.1:

- NEW: Transform4 handle editor
- NEW: Viewer world/projection converter APIs
- NEW: Matrix5x5 inverse, TRS, MultiplyVector/Point4x5
- NEW: Vector5 struct (for Matrix5x5)
- NEW: Global TRS API for Transform4
- NEW: V3Helper support multiple submesh
- NEW: Culling and Better FOV for M4 projection
- NEW: Culling for S4 for better performance
- CHANGE: M4Renderer now always calculate lighting and added VertexProfiles.
- CHANGE: M4Buffer uses V3Helper instead raw mesh to keep data.
- FIX: Small performance optimizations

Jun 4, 2017 – V2.0:

- Full asset rewrite.
- NEW: M4 and S4 scene-wide projection modes.
- CHANGE: Old code now marked as 'Legacy'. Only improvements are made.

Feb 6, 2017 – V1.2:

- NEW: Baking system is now done automatically
- CHANGE: autoUpdate property is now deprecated
- CHANGE: Bake button is now deleted as being unnecessary
- CHANGE: MeshRenderer is now included as component requirement
- FIX: Bug introduced in 1.1.1 where the editor not updating properly

V1.1.1:

- Fix Unity 5.6 Support

V1.1:

- Unity 4.6 Support.
- Faster Baking Time.
- Faster Calculation.
- Fix Morph update at runtime.
- Better Asset Management
- Added Additive RGB Shader (Particles/RGB Cube)

Apr 21, 2015 - V1.0:

- Initial Release.

About

This asset is available in Asset Store published by Wello Soft.

Website | Asset Store | Email