Version 1.5.0

Project Dawn | Dots Plus | Tool

Overview

This package contains essentials needed to develop DOTS projects. It is built to be used as an extension for existing Unity builtin packages (etc. SIMD mathematics, Collections, Jobs and Burst compiler).

Library is developed to fit well with Unity DOTS standards. This includes performance by default, simplicity and most importantly, readability of the code. Driven to take the best HPC# can offer.

Collections

This assembly follows similar standard like unity collection package so it is important to check pages:

- https://docs.unity3d.com/Packages/com.unity.collections@1.3/manual/index.html
- https://docs.unity3d.com/Packages/com.unity.collections@1.3/manual/collection-types.html
- https://docs.unity3d.com/Packages/com.unity.collections@1.3/manual/allocation.html

Native vs Unsafe

The `Native` types perform safety checks to ensure that indexes passed to their methods are in bounds, but the other types in most cases do not. Check page for more information https://docs.unity3d.com/Packages/com.unity.collections@1.3/manual/index.html.

NativeLinkedList/UnsafeLinkedList

An unmanaged, resizable linked list. Linked list is efficient at inserting and removing elements. However, not so efficient with cache usage. Linked list is implemented using double linked nodes, where each node knows its next-node link and previous-node link.

```
var list = new NativeLinkedList<int>(Allocator.Temp);

list.Add(0);
var itr = list.Add(1);
list.Add(2);

list.RemoveAt(itr);

Assert.AreEqual(list.Begin, 0);
Assert.AreEqual(list.Begin.Next, 2);

list.Dispose();
```

> **Note:** Linked list is recommended to be used if you plan to keep added elements iteration position.

NativeStack/UnsafeStack

An managed, resizable stack. Limited version of NativeList that only operates with the last element at the time.

```
var stack = new NativeStack<int>(Allocator.Temp);

stack.Push(0);
stack.Push(1);
stack.Push(2);
stack.Push(3);
Assert.AreEqual(3, stack.Pop());
Assert.AreEqual(2, stack.Pop());
Assert.AreEqual(1, stack.Pop());
Assert.AreEqual(0, stack.Pop());

stack.Dispose();
```

NativePriorityQueue/UnsafePriorityQueue

An unmanaged, resizable priority queue. Priority queue main difference from regular queue that before element enqueue it executes insert sort. It is implemented using linked nodes as a result, inserting does not require pushing data.

```
struct AscendingOrder : IComparer<int>
{
    public int Compare(int x, int y) => x.CompareTo(y);
}

var queue = new NativePriorityQueue<int, AscendingOrder>(Allocator.Temp, new
AscendingOrder());

queue.Enqueue(1);
queue.Enqueue(2);
```

```
Assert.AreEqual(2, queue.Dequeue());
Assert.AreEqual(1, queue.Dequeue());

queue.Dispose();
```

NativeKdTree/UnsafeKdTree

K-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. K-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds. K-d trees are a special case of binary space partitioning trees.

```
struct TreeComparer : IKdTreeComparer<float2>
{
    public int Compare(float2 x, float2 y, int depth)
    {
        int axis = depth % 2;
        return x[axis].CompareTo(y[axis]);
    }

    public float DistanceSq(float2 x, float2 y)
    {
        return math.distancesq(x, y);
    }

    public float DistanceToSplitSq(float2 x, float2 y, int depth)
    {
        int axis = depth % 2;
        return (x[axis] - y[axis]) * (x[axis] - y[axis]);
    }
}

var tree = new NativeKdTree<float2, TreeComparer>(1, Allacator.Temp, new
TreeComparer());

tree.Add(new float2(1, 1));
tree.Add(new float2(2, 2));

Assert.AreEqual(1, FindNearest(new float2(0, 0), out _).Value);

tree.Dispose();
```

> **Note:** K-D tree operations like nearest neighbor search have overhead. As a result, it is recommended to use it when you have lots of points in space.

NativeAABBTree/UnsafeAABBTree

An unmanaged, resizable aabb tree. AABB tree (short for axis aligned bounding box tree) is a space-partitioning data structure for organizing bounding shapes in space. As structure uses generic it is not only usable for boxes, but any shape that implements interfaces. AABB trees are a useful data structure for fast searching bounding shapes in space. AABB trees are a special case of binary space partitioning trees. Based on https://box2d.org/files/ErinCatto_DynamicBVH_GDC2019.pdf.

```
struct AABRectangle : ISurfaceArea<AABRectangle>, IUnion<AABRectangle>,
IOverlap<AABRectangle>
{
    public Rectangle Rectangle;

    public AABRectangle(Rectangle rectangle)
    {
        Rectangle = rectangle;
    }

    public float SurfaceArea() => Rectangle.Perimeter;
    public AABRectangle Union(AABRectangle value) => new
AABRectangle(Rectangle.Union(Rectangle, value.Rectangle));
    public bool Overlap(AABRectangle value) => Rectangle.Overlap(value.Rectangle);
}

...

var tree = new UnsafeAABBTree<AABRectangle>(1, Allocator.Temp);

// Construct tree:
//            N
//          /   \
//         N     N
//        / \   / \
//       a   b c   d

var a = tree.Add(new AABRectangle(new Rectangle(new float2(5, 5), new float2(1,
1))));
var b = tree.Add(new AABRectangle(new Rectangle(new float2(6, 6), new float2(1,
1))));

var c = tree.Add(new AABRectangle(new Rectangle(new float2(-5, -5), new float2(1,
1))));
var d = tree.Add(new AABRectangle(new Rectangle(new float2(-6, -6), new float2(1,
1))));

Assert.AreEqual(tree.Right(tree.Right(tree.Root)), d);
Assert.AreEqual(tree.Left(tree.Right(tree.Root)), c);
Assert.AreEqual(tree.Right(tree.Left(tree.Root)), b);
Assert.AreEqual(tree.Left(tree.Left(tree.Root)), a);

var result = new NativeList<AABRectangle>(2, Allocator.Temp);
tree.FindOverlap(new AABRectangle(new Rectangle(new float2(5, 5), new float2(2,
2))), ref result);
```

```
Assert.AreEqual(2, result.Length);
Assert.IsTrue(tree[a].Rectangle == result[0].Rectangle);
Assert.IsTrue(tree[b].Rectangle == result[1].Rectangle);

result.Dispose();

tree.Dispose();
```

Mathematics

This assembly follows similar standard like unity mathematics package, so it is important to check pages:

- https://docs.unity3d.com/Packages/com.unity.mathematics@1.2/manual/index.html

Why lower case naming convention?

The main reason is that Unity mathematics follows the same conventions.

Only worse thing than bad syntax is inconsistent syntax

Fast Math

There is a small helper class `ProjectDawn.Mathematics.fastmath` that contains several faster math functions at the expense of precision.

Geometry 2D

This assembly provides common 2d shapes used in game development (etc. circle, rectangle, line, capsule, convex polygon...). Those shapes contain useful functions like:

- Testing if shapes `Overlap`.
- Finding minimum `Distance` between shapes.
- Finding `BoundingRectangle`, `InscribedCircle`, `CircumscribedCircle` on the shape.
- Finding `Intersection` of shape and ray.

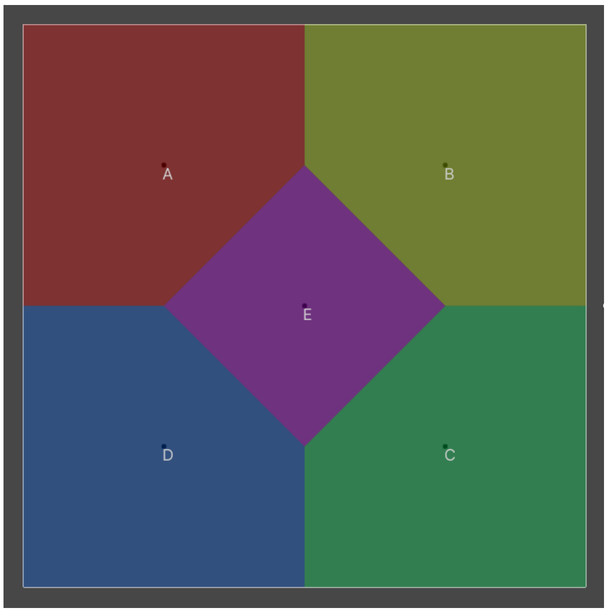
Convex Polygon

There are two APIs for executing convex polygon operations.

- `ConvexPolygon` container that needs to be allocated.
- `ConvexPolygonUtility` static class that can be used with already allocated points array.

Voronoi

Voronoi is a partition of a plane into regions close to each of a given set of objects. In the simplest case, these objects are just finitely many points in the plane (called seeds, sites, or generators). For each seed there is a corresponding region, called a Voronoi cell, consisting of all points of the plane closer to that seed than to any other.



Voronoi generated from 5 sites named A, B, C, D, E.

Voronoi logic is separated into two structures:

- `VoronoiBuilder` - used for gathering sites and constructing voronoi shape.
- `IVoronoiOutput` - the interface used for ouputing voronoi shape.

Voronoi Builder

Structure used for building voronoi. The usage of API is quite simple.

```
// Allocating builder
var builder = new VoronoiBuilder(1, Allocator.Temp);

// Adding site at point 0, 0
builder.Add(new double2(0, 0));

// Outputing the voronoi into `IVoronoiOutput`
builder.Construct(ref output);

// Destroying
builder.Dispose()
```

IVoronoiOutput

This interface is used by `VoronoiBuilder` to call specific callback for constructing voronoi.

```
// Callback after VoronoiBuilder processes the site.
void ProcessSite(double2 point, int index);

// Callback after VoronoiBuilder processes the vertex.
int ProcessVertex(double2 point);
```

```
// Callback after VoronoiBuilder processes the edge.
void ProcessEdge(double a, double b, double c,
int leftVertexIndex, int rightVertexIndex,
int leftSiteIndex, int rightSiteIndex);

// Callback after VoronoiBuilder finished building.
void Build();
```

Here is very simple example that draws wireframe of voronoi.

```
struct CustomVoronoiOutput : IVoronoiOutput
{
    NativeList<float2> m_Vertices;

    public CustomVoronoiOutput(Allocator allocator)
    {
        m_Vertices = new NativeList<float2>(allocator);
    }

    public void Dispose()
    {
        m_Vertices.Dispose();
    }

    public void ProcessSite(double2 point, int siteIndex)
    {
        ShapeGizmos.DrawText((float2)point, $"{(char)(65 + siteIndex)}",
Color.white);
    }

    public void ProcessEdge(double a, double b, double c,
        int leftVertexIndex, int rightVertexIndex,
        int leftSiteIndex, int rightSiteIndex)
    {
        float extent = 120;
        float2 leftVertex = leftVertexIndex != -1 ? m_Vertices[leftVertexIndex] :
new float2(-extent, line((float)a, (float)b, (float)c, -extent));
        float2 rightVertex = rightVertexIndex != -1 ? m_Vertices[rightVertexIndex]
: new float2(extent, line((float)a, (float)b, (float)c, extent));
        ShapeGizmos.DrawLine(leftVertex, rightVertex, Color.green);
    }

    public int ProcessVertex(double2 point)
    {
        ShapeGizmos.DrawSolidCircle((float2)point, 0.1f, Color.green);
        m_Vertices.Add((float2)point);
        return m_Vertices.Length - 1;
    }

    public void Build()
    {
        }
    }
```

VoronoiDiagram

Structure used for constructing cells, edges and vertices that implements `IVoronoiOutput`.

DelaunayTriangulation

Structure used for constructing delaunay dual that implements `IVoronoiOutput`. Quite useful for mesh triangulation.

Geometry 3D

This assembly provides common 3d shapes used in game development (etc. box, sphere, capsule, line, triangle, trinagular surface, ray...). Those shapes contain useful functions like:

- Testing if shapes `Overlap`.
- `Triangle` and `Ray` intersection time.
- `Triangle` and `Triangle` intersection lines.
- `TriangularSurface` similar to mesh.

VertexData/UnsafeVertexData

Structure used for containing vertex data (etc. position, normal, uv ...). Array supports generic data and depends on attributes passed during structure creation. As example it allows having only positions vertex or positions and normals or other setup.

All vertex data is stored in interleaved array. As example VertexData with three attributes - position, normal, uv, would be stored in memory as follows:

a	b	c	d	e	f	...
position0	normal0	uv0	position1	normal1	uv1	...

MeshSurface/UnsafeMeshSurface

Structure equivalent of Unity `Mesh` class. Mainly it has three purposes:

- `Read` - copy `Mesh` into `MeshSurface`.
- `Write` - copy `MeshSurface` into `Mesh`.
- Modify its `VertexData`, `Indices` and `Submeshes`.

Here is small example of creating box `Mesh` using `MeshSurface`:

```
var attributes = new NativeArray<VertexAttributeDescriptor>(1, Allocator.Temp);
attributes[0] = new VertexAttributeDescriptor(VertexAttribute.Position,
VertexAttributeFormat.Float32, 3);
```

```
var surface = new MeshSurface(1, attributes, Allocator.TempJob);

JobHandle dependency = surface.Box(Box, default);

if (TryGetComponent(out MeshFilter meshFilter))
{
    var meshDataArray = Mesh.AllocateWritableMeshData(1);

    dependency = surface.Write(meshDataArray[0], MeshUpdateFlags.Default,
dependency);

    dependency.Complete();

    if (meshFilter.sharedMesh == null)
        meshFilter.sharedMesh = new Mesh();
    Mesh.ApplyAndDisposeWritableMeshData(meshDataArray, meshFilter.sharedMesh);
}

attributes.Dispose();
surface.Dispose();
```

Gizmos

Both 2d and 3d contains helper class `ShapeGizmos` that have methods for drawing shapes in gizmos. It is very useful for debug purpose or drawing gizmos in general.

Here is small example drawing red rectangle:

```
void OnDrawGizmos()
{
    ShapeGizmos.DrawWireRectangle(new Rectangle(new float2(0, 0), new float2(1,
1)), Color.red);
}
```

Tests and Samples

The package contains samples and tests, so it is quite recommended to check them to get an overall sense of API usage.

Dependencies

- Tested with Unity 2020.3
- Package com.unity.mathematics@1.2
- Package com.unity.collections@0.9
- Package com.unity.burst@1.4

Support

If you have questions, bugs or feature requests use Discord.

Current API selection comes from what I personally needed during Unity DOTS project development. This package expected to grow based on the users request. Final goal is to have a library that would make every DOTS developer's life easier (etc. C++ Boost).