

[https://github.com/mezanix/RTS\\_DataStructAlgo.git](https://github.com/mezanix/RTS_DataStructAlgo.git)

# Linked List

# Linked List

A list of nodes each node has 2 fields: data and pointer. The pointer points to another node, next or previous. So linked list can be single way, double way, circular or even spatial.

[Link](#)

Challenge:

Implement a generic linked list, means the data is a generic type. So you can create this list with any type you want.


Implementation consist of creating a node class and a linked list class collecting the nodes, as well as the ability to insert after/before nodes and remove nodes.

Nice application of the challenge:

Instantiate a linked list of type Action, so you can create a list of actions, means a precise ordered list of commands.




# Linked List

Node:

 Data held by the node

 Next node

 Previous node

```
public class IntLinkedListNode
{
     public int data = 0;
     public IntLinkedListNode next = null;
     public IntLinkedListNode previous = null;


    2 references
    public IntLinkedListNode(int data, IntLink
    {
        this.data = data;
        this.next = next;
        this.previous = previous;
    }
}
```


# Linked List

List:

 First node

 Current node

```
public class IntLinkedList
{
     IntLinkedListNode first = null;

     public IntLinkedListNode current = null;


    1 reference
    public IntLinkedList(IntLinkedListNode fir
```


# Linked List

Travel Forward:

 Check if last node


 Travel

```
public void Travel()
{
     if (current.next == null)
    {
        return;
    }



     current = current.next;
}
```

# Linked List

Travel Back:

 Check if first node

 Travel

```
public void TravelBack()  
{  
     if (current.previous == null)  
        return;  
  
     current = current.previous;  
}
```

# Binary Tree



# Binary Tree

A list of nodes each node has 3 fields: data and 2 pointers. The pointers point to another nodes, left and right. Binary Tree makes searching data very fast. The speed of search come from the structured nature of the data.

[How structured data is faster for search](#)

[How to traverse breadth first and depth first](#)

Challenge:

Implement a int binary tree with the rules smaller to the left and bigger to the right.

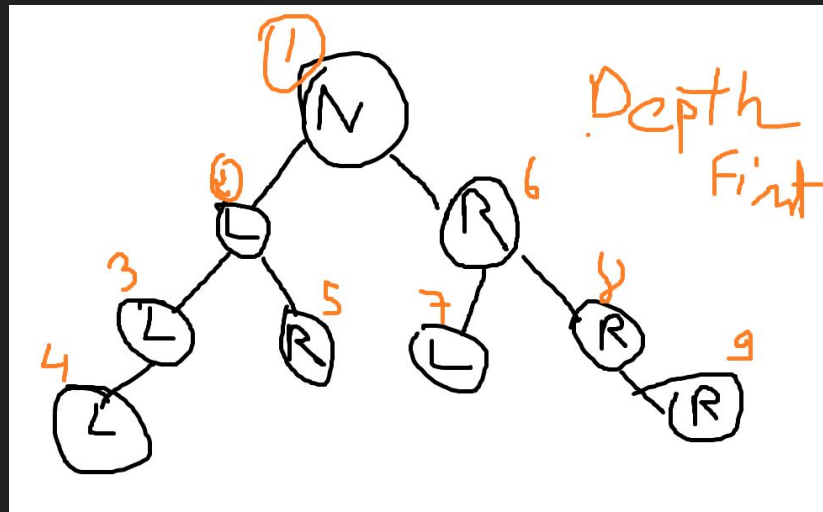
Implement as well a traverse breadth first and a travel depth first methods.

Search a number in the tree and count the number of iterations consumed to find the number.

# Binary Tree

Depth first:

- Is attempt possible
- Attempt to go left
- Attempt to go right



```
public static void DepthFirstTravel<B>(BinaryTreeNode<B> node, Action<B> action)
{
    ● if (node == null)
        return;

    ● action(node.data);

    ● DepthFirstTravel(node.left, action);
    ● DepthFirstTravel(node.right, action);
}
```

# Binary Tree

Breadth first:

- Process and dequeue
- Enqueue left
- Enqueue right
- Repeat while queue has items

```
public static void BreadthFirstTravel<B>(BinaryTreeNode<B> node, Action<B> action)
{
    Queue<BinaryTreeNode<B>> queue = new Queue<BinaryTreeNode<B>>();

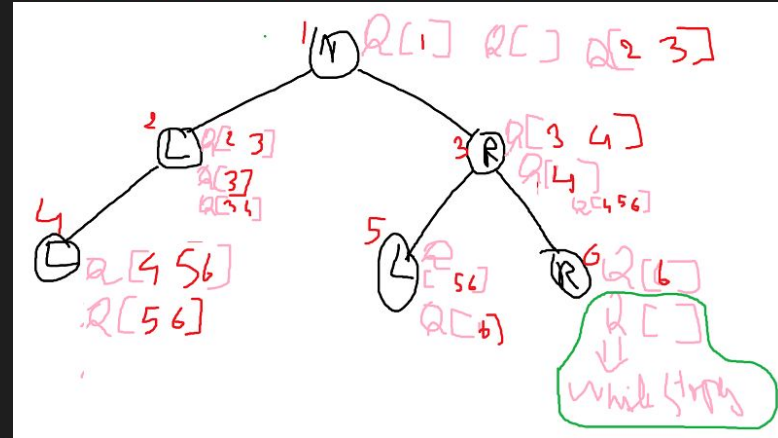
    queue.Enqueue(node);

    while(queue.Count > 0)
    {
        node = queue.Dequeue();

        action(node.data);

        if(node.left != null)
        {
            queue.Enqueue(node.left);
        }

        if(node.right != null)
        {
            queue.Enqueue(node.right);
        }
    }
}
```

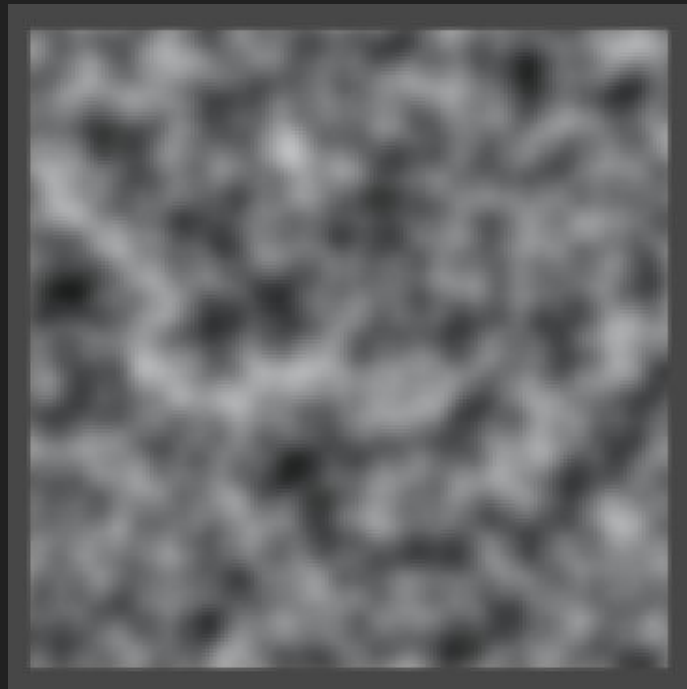


# Perlin Noise

# Perlin Noise

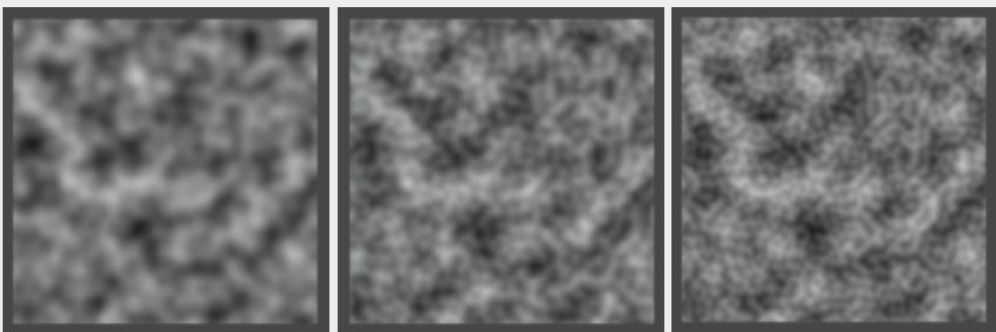
A [gradient fractal](#) noise, procedurally generates organic like textures.

Perlin build from scratch with [CatLikeCoding](#).

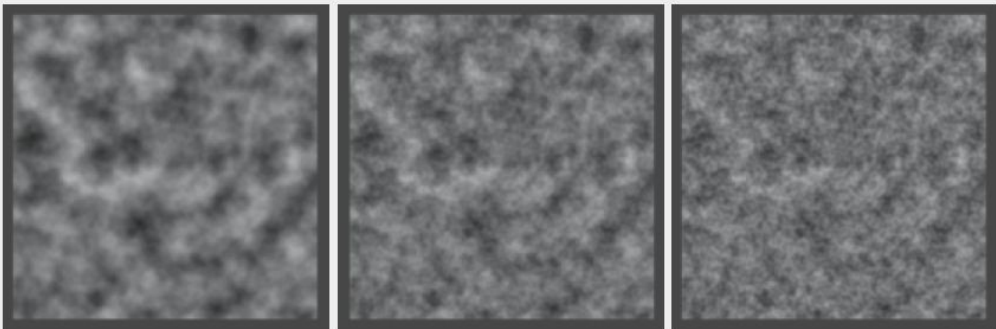


# Perlin Noise

In our code we did not build it from scratch instead we used [Sebastian Lague](#) implementation. Where we apply a fractal logic on a Unity built-in simple Perlin noise:



Two octaves of Perlin noise with base frequency 8, using lacunarity 2, 3, and 4.



Three octaves of Perlin noise with base frequency 8, using persistence  $\frac{1}{2}$ ,  $\frac{3}{4}$ , and 1.

```
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        float amplitude = 1f;
        float frequency = 1f;
        float noiseHeight = 0f;

        for (int o = 0; o < octaves; o++)
        {
            float sampleX = x / scale * frequency;
            float sampleY = y / scale * frequency;

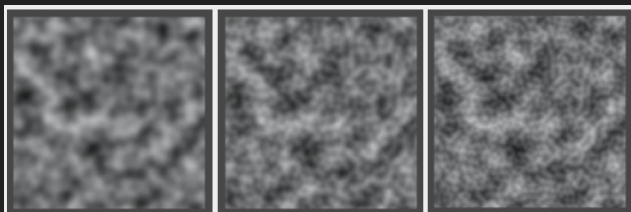
            float perlinValue = Mathf.PerlinNoise(
                sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistence;
            frequency *= lacunarity;
        }
    }
}
```

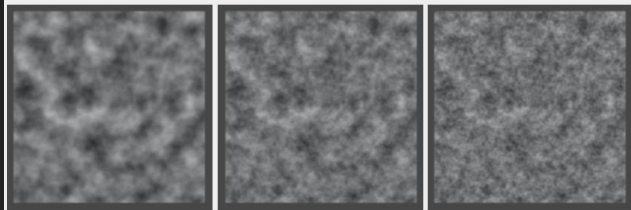
# Perlin Noise: Assets\Lib\Scripts\Noise

- Octaves: Number of loops to achieve a fractal effect.
- Persistence: Amount of “surviving” of higher frequencies inside higher octaves.
- Lacunarity: Amount of frequency growth for higher octaves.

Higher frequency means smaller details.



Two octaves of Perlin noise with base frequency 8, using lacunarity 2, 3, and 4.



Three octaves of Perlin noise with base frequency 8, using persistence 1/2, 3/4, and 1.

```
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        float amplitude = 1f;
        float frequency = 1f;
        float noiseHeight = 0f;

        for (int o = 0; o < octaves; o++)
        {
            float sampleX = x / scale * frequency;
            float sampleY = y / scale * frequency;

            float perlinValue = Mathf.PerlinNoise(
                sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistence;
            frequency *= lacunarity;
        }
    }
}
```

# Pathfinding



# Breadth first search

- Frontier
- Reached
- Start
- Target

```
public IEnumerator BreadthFirstTravel(GridCell2D start)
{
    //this.start = start;

    Queue<GridCell2D> frontier = new Queue<GridCell2D>();
    HashSet<GridCell2D> reached = new HashSet<GridCell2D>();

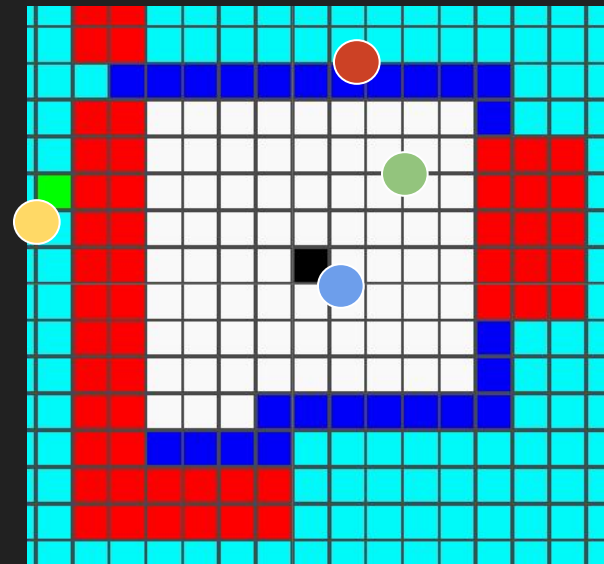
    frontier.Enqueue(start);
    reached.Add(start);

    while (frontier.Count > 0)
    {
        GridCell2D current = frontier.Dequeue();
        foreach (GridCell2D t in current.Neighbors)
        {
            if (reached.Contains(t))
            {
                continue;
            }

            frontier.Enqueue(t);
            reached.Add(t);
        }

        foreach (GridCell2D t in reached) ...
        foreach (GridCell2D t in frontier) ...

        yield return new WaitForSeconds(animationWait);
    }
}
```



# Breadth first search (Dijkstra)

- Enqueue Start into Frontier
- Add Start to Reached
- Repeat the addings
- For each cell neighbors
- If the neighbor is not reached
- Dequeue each Frontier before processing it

```
public IEnumerable BreadthFirstTravel(GridCell2D start)
{
    //this.start = start;

    Queue<GridCell2D> frontier = new Queue<GridCell2D>();
    HashSet<GridCell2D> reached = new HashSet<GridCell2D>();

    frontier.Enqueue(start);
    reached.Add(start);

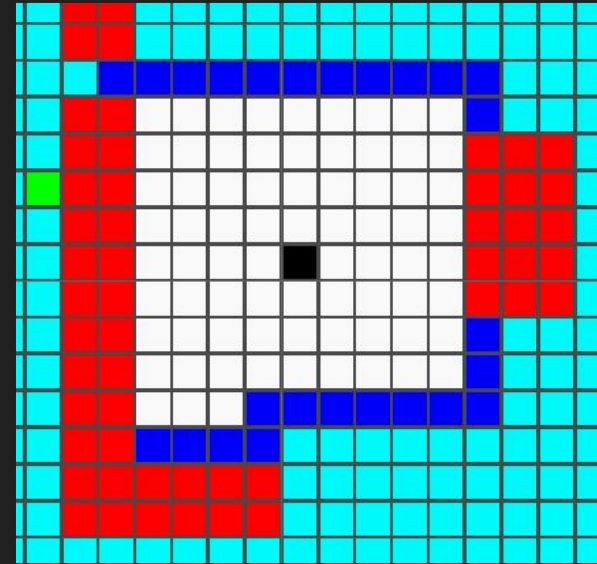
    while (frontier.Count > 0)
    {
        GridCell2D current = frontier.Dequeue();

        foreach (GridCell2D t in current.Neighbors)
        {
            if (reached.Contains(t))
                continue;

            frontier.Enqueue(t);
            reached.Add(t);
        }

        foreach (GridCell2D t in reached) ...
        foreach (GridCell2D t in frontier) ...

        yield return new WaitForSeconds(animationWait);
    }
}
```



# Breadth first search with path track

- Tracked path
- Tracked path dictionary
- Coming from current
- Going to neighbor
- All the dictionary couples  
(going to, coming from)  
Together construct the path.
- Prevent walls
- Early exit, target found.

```
public IEnumerator BreadthFirstTravelPathTrack()
{
    Queue<GridCell2D> frontier = new Queue<GridCell2D>();
    cameFrom.Clear();

    frontier.Enqueue(grid.StartCell);
    cameFrom.Add(grid.StartCell, null);

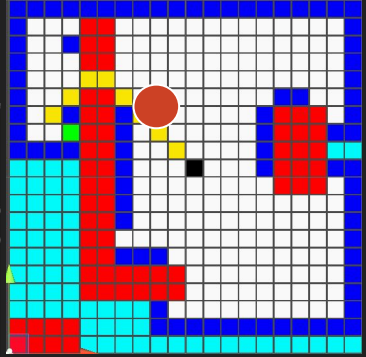
    while (frontier.Count > 0)
    {
        GridCell2D current = frontier.Dequeue();
        foreach (GridCell2D t in current.Neighbors)
        {
            if (t.IsWall())
                continue;

            if (cameFrom.ContainsKey(t))
                continue;

            frontier.Enqueue(t);
            cameFrom.Add(t, current);
        }

        if (current == grid.TargetCell)
            break;
    }
}
```

// key: cell going to  
// value: cell coming from  
Dictionary<GridCell2D, GridCell2D> cameFrom = new Dictionary<GridCell2D, GridCell2D>();



# Breadth first search with Cost

● Costly areas

● CostSoFar variable

● Current added to next cost

● New cost assigned to the next cell

Cell after cell the cost grow, the high costly areas will raise faster.

● Path prevents costly area

● By prioritizing less costly areas

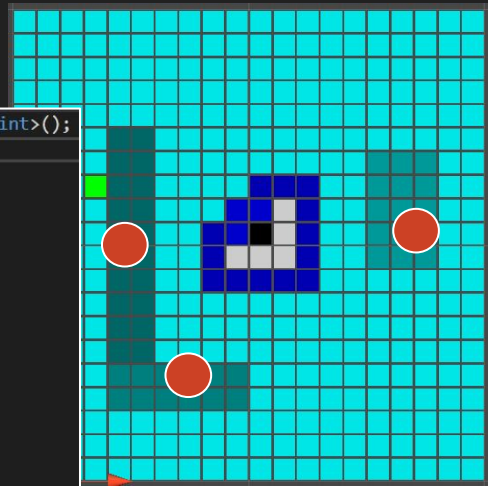
```
Dictionary<GridCell2D, int> costSoFar = new Dictionary<GridCell2D, int>();
costSoFar.Add(grid.StartCell, grid.StartCell.Cost);

while (frontier.Count > 0)
{
    GridCell2D current = frontier[frontier.Count - 1];
    current.SetColor(GridCell2D.visitedColor);
    frontier.RemoveAt(frontier.Count - 1);

    foreach (GridCell2D next in current.Neighbors)
    {
        if (next.IsWall())
            continue;

        int newCost = costSoFar[current] + next.Cost;

        if (costSoFar.ContainsKey(next) == false ||
            newCost < costSoFar[next])
        {
            if (costSoFar.ContainsKey(next) == false)
            {
                costSoFar.Add(next, newCost);
                next.Cost = newCost;
            }
            else
            {
                costSoFar[next] = newCost;
                next.Cost = newCost;
            }
        }
    }
}
```



# Breadth first search, greedy first, prioritize smaller distance

- Sort frontier after each add, sorting is by smaller distance
- Each added cell knows its distance
- Dequeue the last cell from the sorted frontier before using it



```
frontier.Add(grid.StartCell);
frontier.Sort(new GridCell2DDistanceToTargetComparer());

cameFrom.Add(grid.StartCell, null);

while (frontier.Count > 0)
{
    GridCell2D current = frontier[frontier.Count - 1];
    current.SetColor(GridCell2D.visitedColor);
    frontier.RemoveAt(frontier.Count - 1);

    foreach (GridCell2D next in current.Neighbors)
    {
        if (next.IsWall())
            continue;

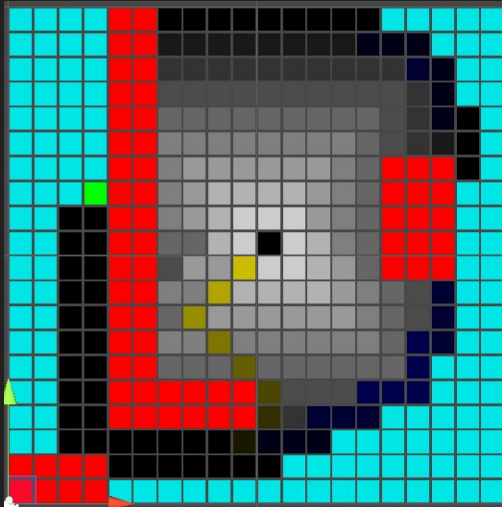
        if (cameFrom.ContainsKey(next))
            continue;

        if (next == grid.TargetCell)
        {
            next.DistanceToTarget = ManhattanDistance(next.Index, grid.TargetCell.Index);
            frontier.Add(next);
            frontier.Sort(new GridCell2DDistanceToTargetComparer());
        }
    }
}
```



# Breadth first search, A\*, Combines cost and small distance

- Sort frontier after each add, sorting is by smaller distance and cost
- Each added cell knows its distance and cost
- Dequeue the last cell from the sorted frontier before using it



```
frontier.Add(grid.StartCell);
frontier.Sort(new GridCell2DDistanceToTargetAndCostComparer());
cameFrom.Add(grid.StartCell, null);
Dictionary<GridCell2D, int> costSoFar = new Dictionary<GridCell2D, int>();
costSoFar.Add(grid.StartCell, grid.StartCell.Cost);

while (frontier.Count > 0)
{
    GridCell2D current = frontier[frontier.Count - 1];
    current.SetColor(GridCell2D.visitedColor);
    frontier.RemoveAt(frontier.Count - 1);

    foreach (GridCell2D next in current.Neighbors)
    {
        if (next.IsWall())
            continue;

        int newCost = costSoFar[current] + next.Cost;

        if (costSoFar.ContainsKey(next) == false ||
            newCost < costSoFar[next])
        {
            if (costSoFar.ContainsKey(next) == false)
            {
                costSoFar.Add(next, newCost);
                next.Cost = newCost;
            }
            else
            {
                costSoFar[next] = newCost;
                next.Cost = newCost;
            }

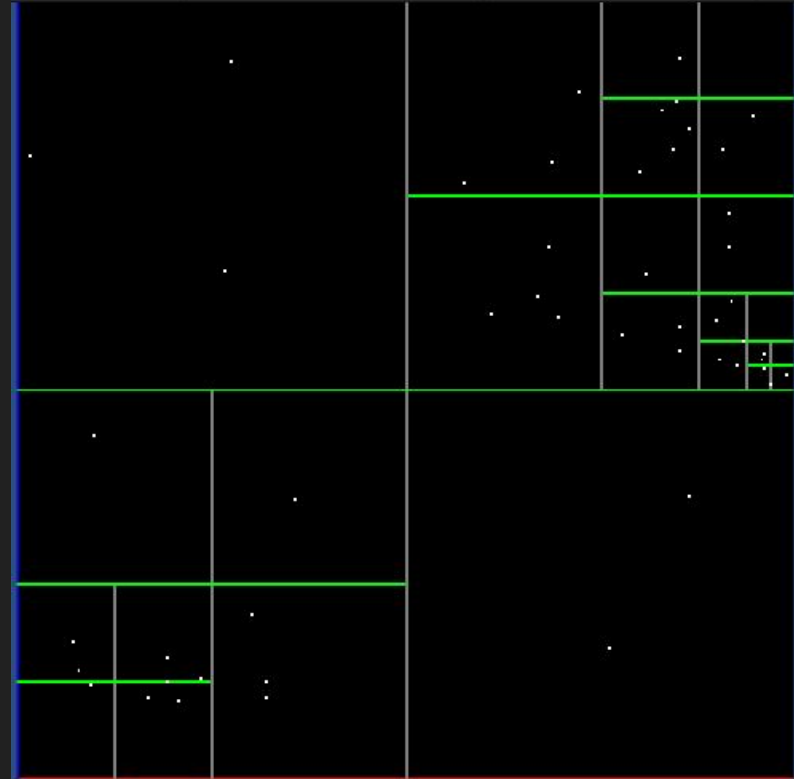
            if (next == grid.TargetCell)
            {
                next.DistanceToTarget = ManhattanDistance(next.Index, grid.TargetCell.Index);
                frontier.Add(next);
                frontier.Sort(new GridCell2DDistanceToTargetAndCostComparer());
            }
        }
    }
}
```

# Quadtree

# Quadtree

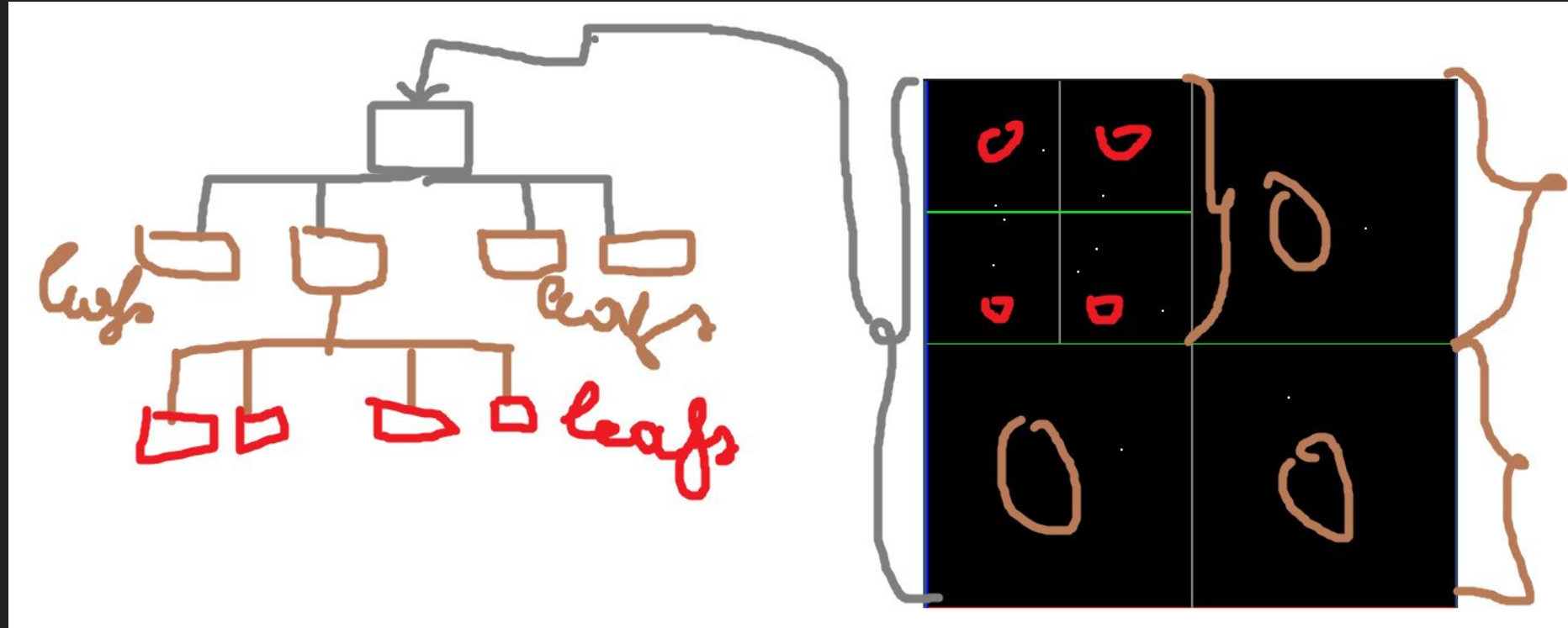
A spatial partitioning of space. A criteria guide the partition.

For example: more points leads to more partitions.



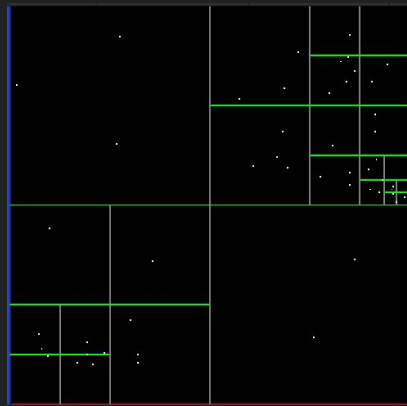


# Quadtree, visualization



# Quadtree

- Not full yet, dividing criteria not met
- Dividing criteria met
- Subdivide
- Transfer the data or actions to the new quads



```
public bool Insert(Point point)
{
    if (boundary.Contains(point) == false)
    {
        return false;
    }

    if (points.Count < capacity)
    {
        points.Add(point);
        addedBeforeDivide.Add(point);
        return true;
    }
    else if (points.Count >= capacity)
    {
        if (divided == false)
        {
            points.Clear();
            Subdivide();
        }

        if (northEast.Insert(point) == true)
        else if (northWest.Insert(point) == true)
        else if (southEast.Insert(point) == true)
        else if (southWest.Insert(point) == true)
        {
            addedBeforeDivide.Clear();
        }

        //error
        return false;
    }
}
```

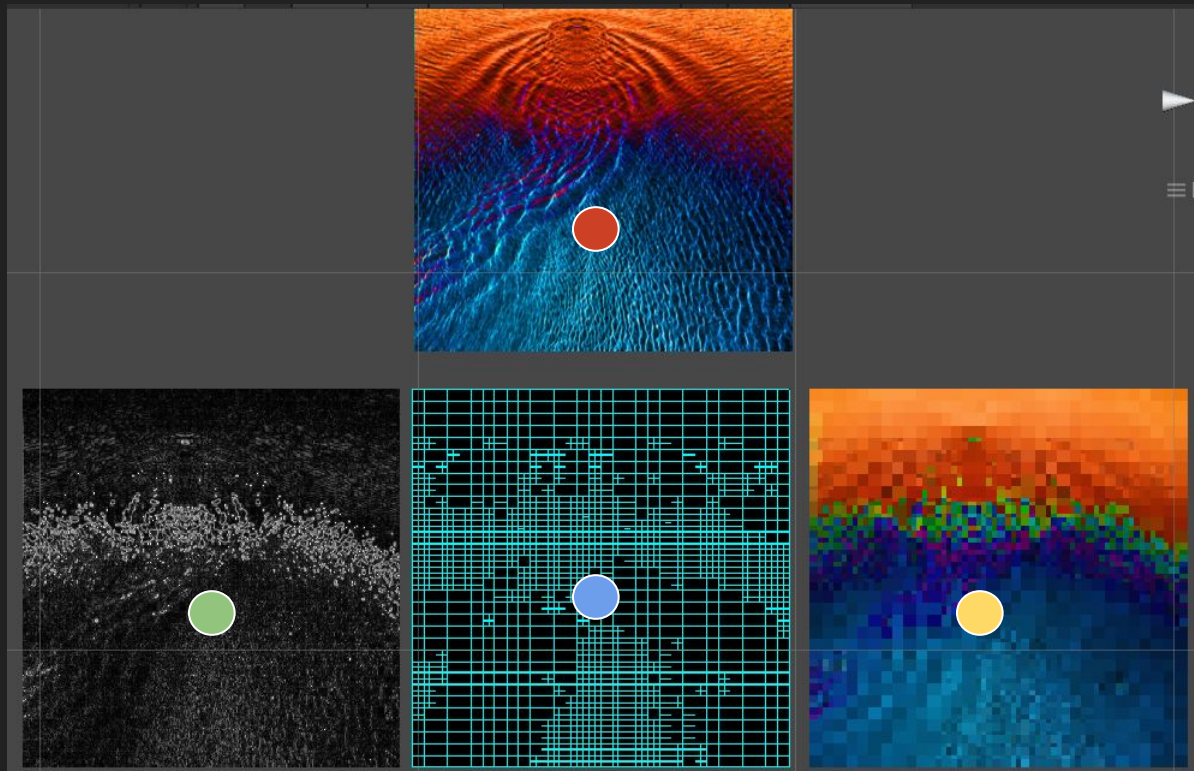
# Quadtree, Image compression

# Quadtree, Image compression

- Source Image
- Map of color changes
- Quadtree map
- Compressed image using source and quadtree

Note:  
Press space to generate the  
quadtree.

Press G to colorize.



# Quadtree, map of color changes, Laplacian

- Source Image
- Map of color changes
- Laplacian

N	N	N
N	C	N
N	N	N

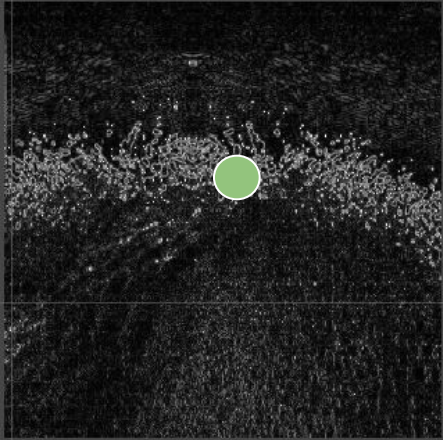
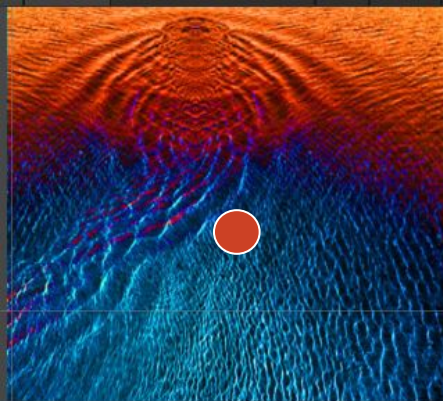
● Laplacian =

$$|C - \frac{\sum N}{8}|$$

↑  
Central value

↑  
Average of neighbors

How much C deviates from neighbors average



# Quadtree, map of color changes, Laplacian

● Subtraction

● Central value

● Average

```
public static Vector3 AverageOfHSV(this Texture2D t, int
```

```
{  
    Color[] neibs = new Color[8];
```

```
    neibs[0] = t.GetPixel(x - step, y - step);
```

```
    neibs[1] = t.GetPixel(x - step, y);
```

```
    neibs[2] = t.GetPixel(x - step, y + step);
```

```
    neibs[3] = t.GetPixel(x, y + 1);
```

```
    neibs[4] = t.GetPixel(x + step, y - step);
```

```
    neibs[5] = t.GetPixel(x + step, y);
```

```
    neibs[6] = t.GetPixel(x + step, y + step);
```

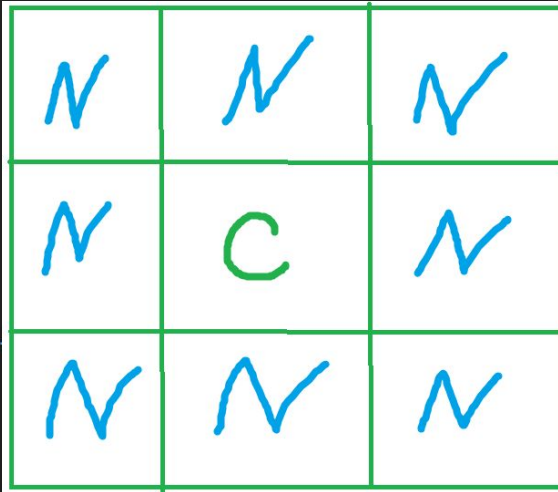
```
    neibs[7] = t.GetPixel(x, y - step);
```

```
    return new Vector3(  
        HueAverage(neibs),
```

```
        SaturationAverage(neibs),
```

```
        ValueAverage(neibs));
```

```
}
```



Laplacian =

$$|C - \frac{\sum N}{8}|$$

↓ Central value      ↓ Average of neighbors

How much C deviates from neighbors average

```
private float LaplacianOnHSV(int x, int y)
```

```
{
```

```
    return Vector3.Distance(texture.GetPixel(x, y).GetHsv(), texture.AverageOfHSV(x, y, step));
```

```
}
```



# Quadtree, quads following the changes map

● Map of color changes

● Quadtree

● Is there enough changes to subdivide

```
public void Subdivide()
{
    if (divided == true)
        return;

    if (RelativeCumulatedWhite() < relativeLimit)
        return;

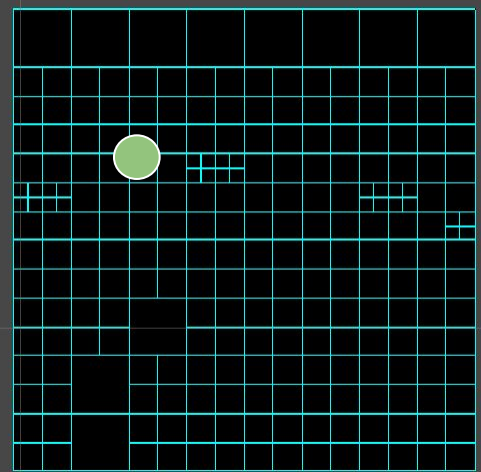
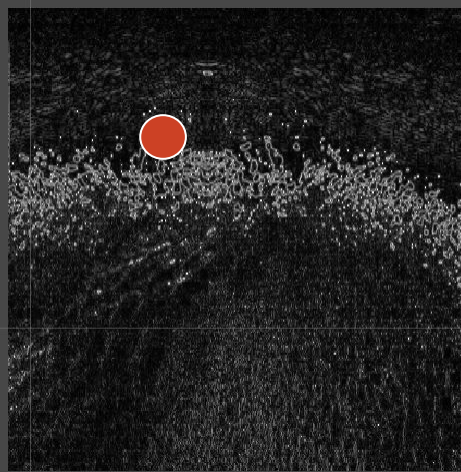
    DoSubdivide();
}

1 reference
void DoSubdivide()
{
    northEast = new QuadtreeTextureCompression(boundary.NorthEast, map, sourceImage);
    northWest = new QuadtreeTextureCompression(boundary.NorthWest, map, sourceImage);

    southEast = new QuadtreeTextureCompression(boundary.SouthEast, map, sourceImage);
    southWest = new QuadtreeTextureCompression(boundary.SouthWest, map, sourceImage);

    divided = true;

    northEast.Subdivide();
    northWest.Subdivide();
    southEast.Subdivide();
    southWest.Subdivide();
}
```



# Quadtree, changes map: relative cumulated white

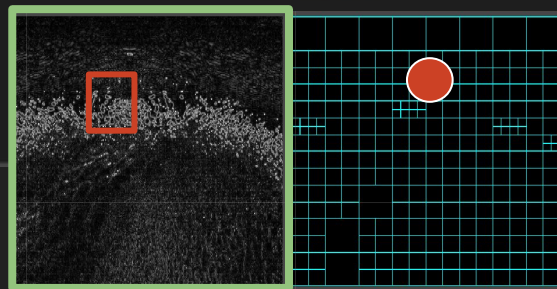
● Boundary quad  
cumulating white pixels

● Total texture amount of  
white

Note: white pixel = color  
changes

```
float RelativeCumulatedWhite()
{
    ●
    ●
    ...return PartialCumulatedWhite(map) / maxCumulatedWhite;
}

1 reference
float PartialCumulatedWhite(Texture2D texture)
{
    ...float r = 0f;
    ...for (float y = boundary.South; y < boundary.North; y += 1f)
    ...{
    ...    for (float x = boundary.West; x < boundary.East; x += 1f)
    ...    {
    ...        ...r += texture.GetPixel((int)x, (int)y).r;
    ...    }
    ...}
    ...return r;
}
```



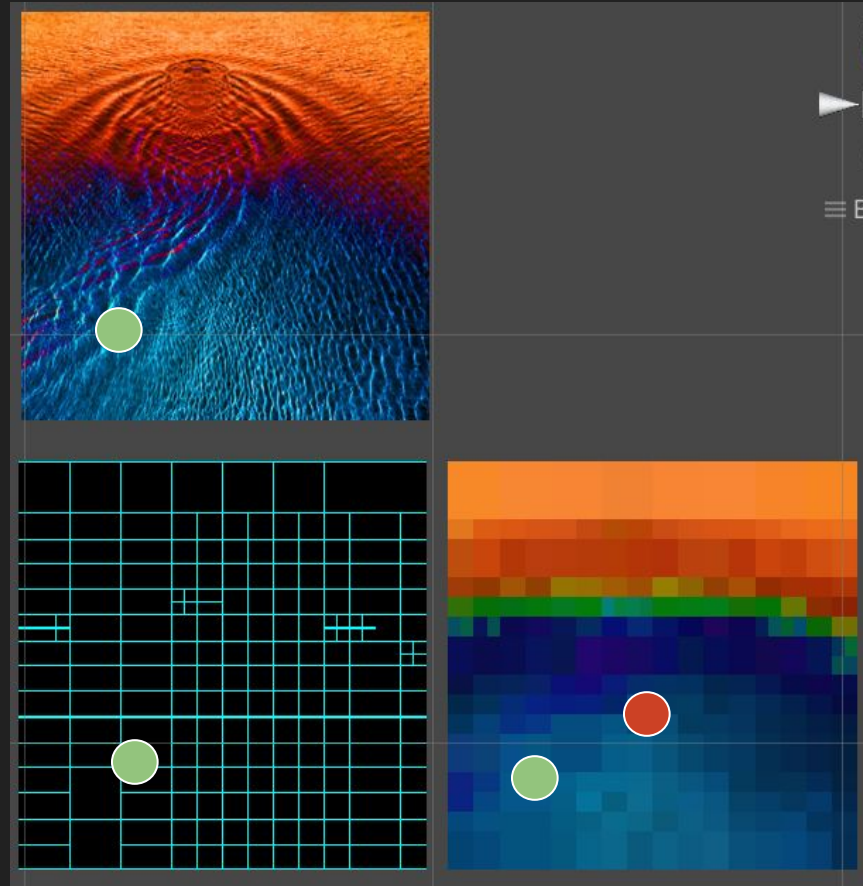


# Quadtree, Colorize the compressed texture

- Create a texture with resolution according to the smallest quadtree quad.
- Find normalized (UV) position of pixels, red point is  $\frac{1}{4} = 0.25$  of the size.  
A normalized -universal- position is a common ground between spaces of different resolutions.

After spotting the right position, detect the dominant color in this quad:

- Central point as dominant, or
- Average One



# Quadtree, Colorize the compressed texture, the method

● Source Texture

● Texture to colorize

● UV, normalized universal position

● Divide by size

● Multiply by size

```
1 reference
public static Vector2 PixelPositionToUv(this Texture2D t, int x, int y)
{
    return new Vector2(
        (float)x / (float)(t.width - 1),
        (float)y / (float)(t.height - 1));
}
```

```
1 reference
public static Vector2Int UvToPixelPosition(this Texture2D t, float x, float y)
{
    return new Vector2Int(
        Mathf.RoundToInt(x * (t.width - 1)),
        Mathf.RoundToInt(y * (t.height - 1)));
}
```

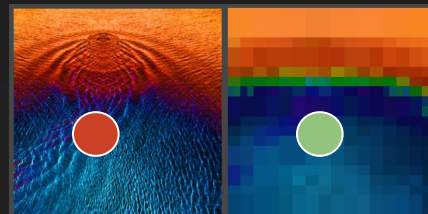
```
1 reference
public void Colorize(Texture2D texture)
{
    List<QuadtreeTextureCompression> leafs = new List<QuadtreeTextureCompression>();
    GetLeafs(leafs);

    for (int y = 0; y < texture.height; y++)
    {
        for (int x = 0; x < texture.width; x++)
        {
            Vector2 uv = texture.PixelPositionToUv(x, y);

            Vector2Int positionInSource = sourceImage.UvToPixelPosition(uv.x, uv.y);

            for (int i = 0; i < leafs.Count; i++)
            {
                if (leafs[i].boundary.Contains(
                    new Point((float)positionInSource.x,
                        (float)positionInSource.y)) == false)
                {
                    continue;

                    texture.SetPixel(x, y, leafs[i].GetDominantColorFromSource());
                    break;
                }
            }
        }
    }
}
```



# Quadtree, Boundary -Quad- GetDominantColor

● Get all pixels of the boundary  
-Quad-

● Average them

OR

● Use only the central value of  
the quad

● Using average: accurate  
about gradient.

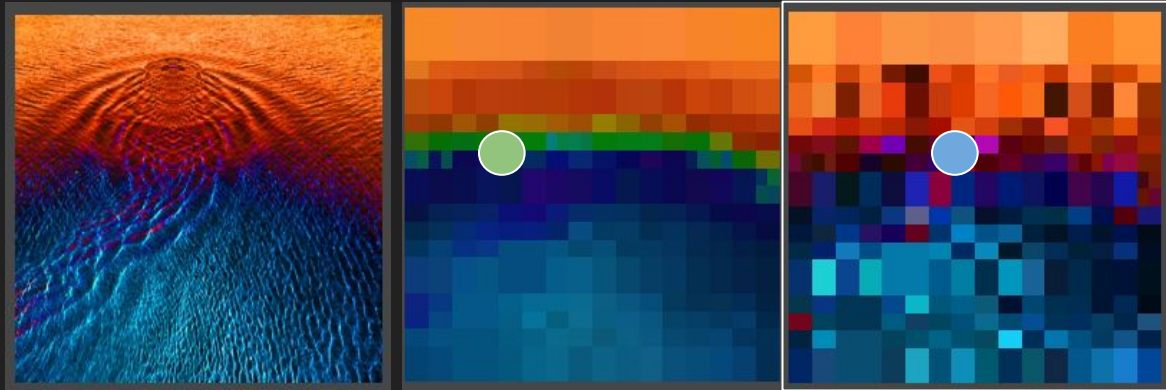
● Using central: accurate about  
color

```
private Color GetDominantColorFromSource()
{
    Color[] colors =
        sourceImage.GetPixels(
            (int)boundary.West, (int)boundary.South,
            (int)boundary.DoubleWidth, (int)boundary.DoubleHeight);

    Vector3 hsvAverage = new Vector3(
        TextureExtensions.HueAverage(colors),
        TextureExtensions.SaturationAverage(colors),
        TextureExtensions.ValueAverage(colors));

    return Color.HSVToRGB(hsvAverage.x, hsvAverage.y, hsvAverage.z);

    //return sourceImage.GetPixel((int)boundary.centerX, (int)boundary.centerY);
}
```

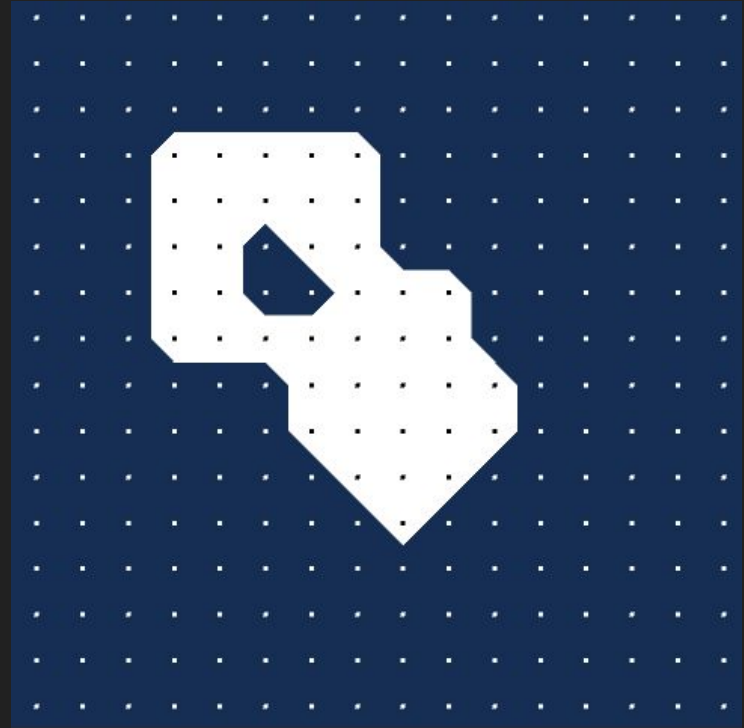


# Marching Squares

# Marching Squares

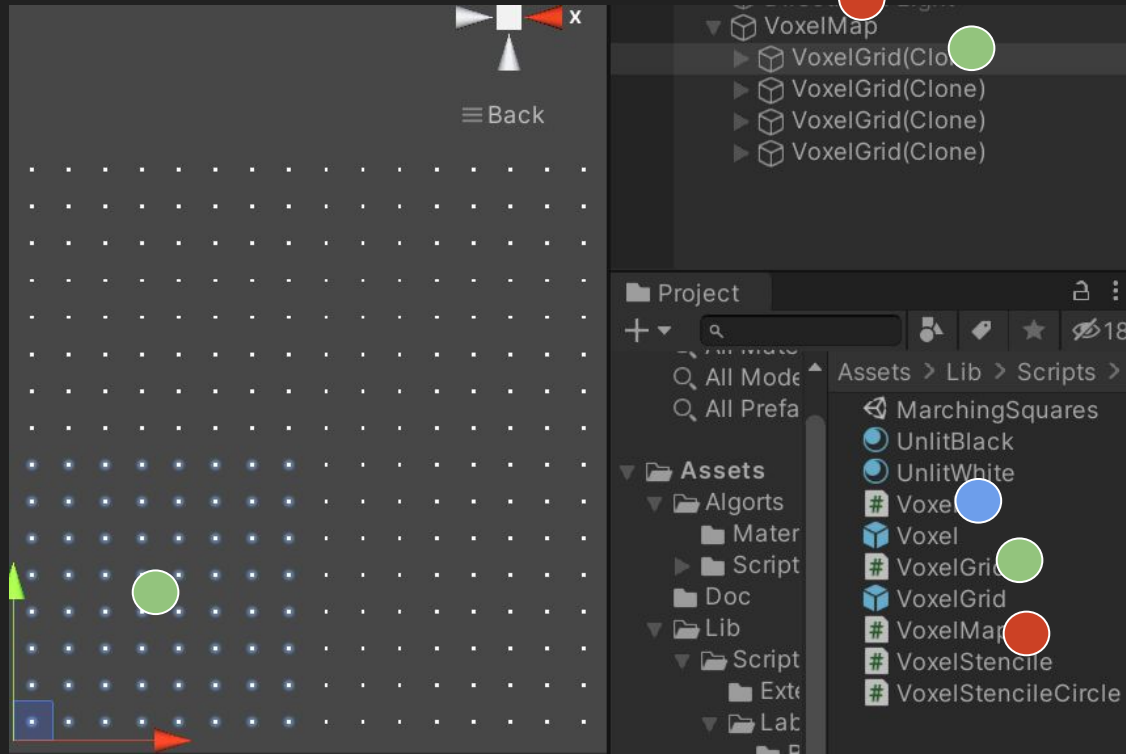
Marching squares generates contours for a cloud of 2D points

<https://catlikecoding.com/unity/tutorials/marching-squares/>



# Marching Squares, Map and Chunks

We organize the points -voxels- cloud inside a grid -chunk- and chunks form a chunk grid inside a map.





# Marching Squares, VoxelMap Class

- Deal with inputs
- Generates the chunks -voxelGrids-
- Edit voxels -turning them on or off- according to inputs -clicks-
- Generated chunks form a 2D pattern

```
private void InteractInput()...
```

1 reference

```
private void EditVoxels(Vector3 point)...
```

1 reference

```
private void CreateChunk(int i, int x, int y)...
```

```
private void Awake()
```

```
{
```

```
    BoxCollider collider = gameObject.AddComponent<BoxCollider>();
```

```
    collider.size = new Vector3(size, size, 1f);
```

```
    halfSize = size * 0.5f;
```

```
    chunkSize = size / chunkResolution;
```

```
    voxelSize = chunkSize / voxelResolution;
```

```
    chunks = new VoxelGrid[chunkResolution * chunkResolution];
```

```
    for (int y = 0; y < chunkResolution; y++)
```

```
    {
```

```
        for (int x = 0; x < chunkResolution; x++, i++)
```

```
        {
```

```
            CreateChunk(i, x, y);
```

```
        }
```

```
    }
```

```
}
```

# Marching Squares, VoxelGrid Class

● Triangulate to create the contour

● Set voxel color to know which is active

● Create Voxels

● Apply the user stencil, to activate or not the voxels

```
public void Apply(VoxelStencile stencile)
{
    for (int y = stencile.yStart; y <= stencile.yEnd; y++)
    {
        for (int i = y * resolution + stencile.xStart;
            for (int x = stencile.xStart; x <= stencile.xEnd; x++, i++)
            {
                voxels[i].state = stencile.Apply(x, y, voxels[i].state);
            }
        }
    }
    Refresh();
}
```

```
public void Init(int resolution, float size)...
```

2 references

```
private void Refresh()...
```

1 reference

```
private void Triangulate()...
```

1 reference

```
private void TriangulateCellRows()...
```

1 reference

```
private void triangulateGapCell(int i)...
```

1 reference

```
void TriangulateGapRow()...
```

4 references

```
private void TriangulateCell(Voxel a, Voxel b, Voxel c, Voxel d)...
```

4 references

```
private void AddPentagon(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, Vector3 v4)...
```

5 references

```
private void AddQuad(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3)...
```

8 references

```
private void Addtriangle(Vector3 v_0, Vector3 v_1, Vector3 v_2)...
```

1 reference

```
private void SetVoxelColors()...
```

1 reference

```
private void CreateVoxel(int i, int x, int y)...
```

1 reference

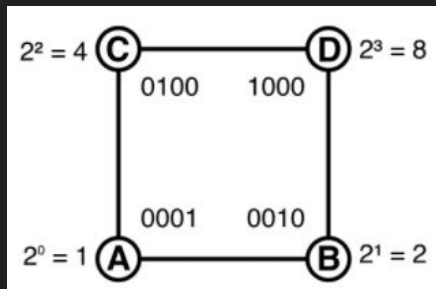
```
public void Apply(VoxelStencile stencile)...
```



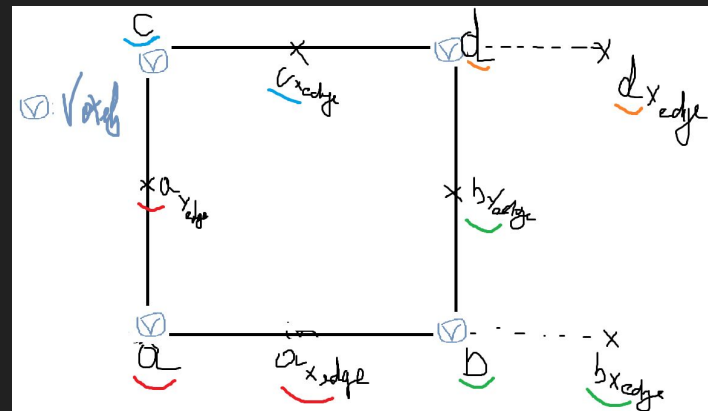
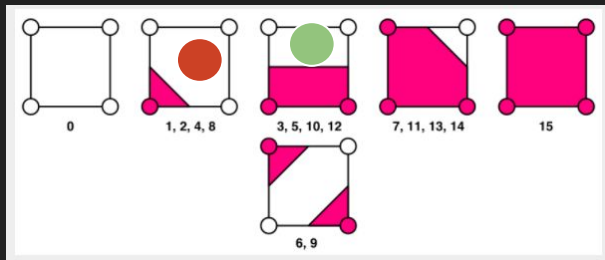
# Marching Squares, VoxelGrid Class, TriangulateCell

- 1: 0001 a is active
- 2: 0010 b is active
- 4: 0100 c is active
- 8: 1000 d is active

Other cases are combinations of the previous ones:  
Example:  
3: 0011 a and b are active.



```
private void TriangulateCell(Voxel a, Voxel b, Voxel c, Voxel d)
{
    int cellType = 0;
    if (a.state)
        cellType |= 1;
    if (b.state)
        cellType |= 2;
    if (c.state)
        cellType |= 4;
    if (d.state)
        cellType |= 8;
    switch (cellType) { ... }
}
```



# Marching Squares, VoxelGrid Class, TriangulateCell, other cases

Other cases are combinations of the previous ones:

Example:

3: 0011 a and b are active.

```
switch(cellType)
{
    case 0:
        return;

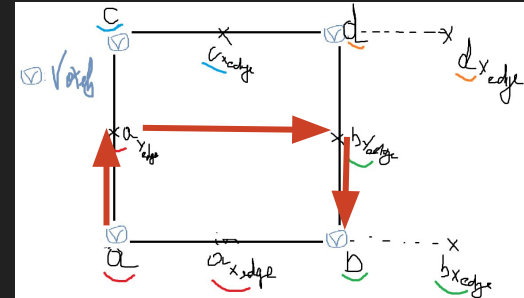
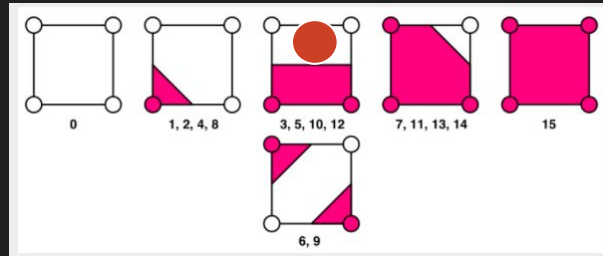
    case 1:
        Addtriangle(a.position, a.y_edgePosition, a.x_edgePosition);
        break;

    case 2:
        Addtriangle(b.position, a.x_edgePosition, b.y_edgePosition);
        break;

    case 4:
        Addtriangle(a.y_edgePosition, c.position, c.x_edgePosition);
        break;

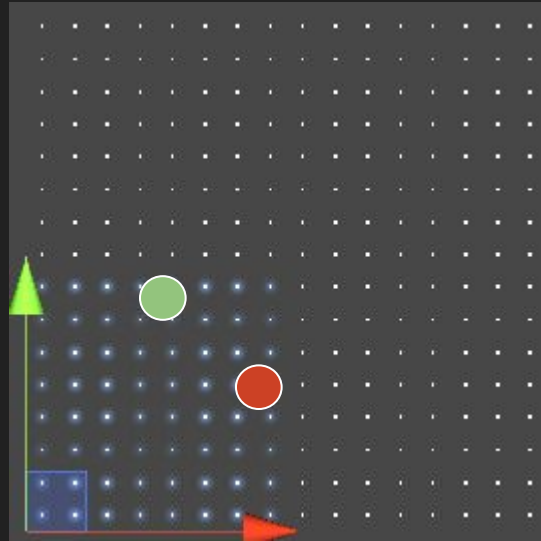
    case 8:
        Addtriangle(c.x_edgePosition, d.position, b.y_edgePosition);
        break;

    case 3:
        AddQuad(a.position, a.y_edgePosition, b.y_edgePosition, b.position);
        break;
}
```



# Marching Squares, VoxelGrid Class, Triangulate, details

- If a X neighbor grid exists:  
Prepare to bridge with it.
- If a Y neighbor grid exists:  
Bridge with it.
- Triangulates the inside voxel grid and the gap with the X neighbors



```
private void Triangulate()
{
    vertices.Clear();
    triangles.Clear();
    mesh.Clear();

    if (x_neib != null)
    {
        {
            dummy_x.Become_X_DummyOf(x_neib.voxels[0], gridSize);
        }
    }

    TriangulateCellRows();

    if (y_neib != null)
    {
        TriangulateGapRow()
    }

    mesh.vertices = vertices.ToArray();
    mesh.triangles = triangles.ToArray();
}
```

# Marching Squares, VoxelGrid Class, Triangulate, Details

● Inside the grid

● The gap with the X neighbors

```
public class Voxel
{
    ... public bool state = false;
    ... public Vector3 position = Vector3.zero;
    ... public Vector3 x_edgePosition = Vector3.zero;
    ... public Vector3 y_edgePosition = Vector3.zero;

    3 references
    ... public Voxel()...

    1 reference
    ... public Voxel(int x, int y, float size)...

    2 references
    ... public void Become_X_DummyOf(Voxel voxel, float offset)...

    2 references
    ... public void Become_Y_DummyOf(Voxel voxel, float offset)...

    1 reference
    ... public void Become_XY_DummyOf(Voxel voxel, float offset)...
}
```

```
private void TriangulateCellRows()
{
    ... int cells = resolution - 1;
    ... for (int y = 0, i = 0; y < cells; y++, i++)
    ... {
    ...     for (int x = 0; x < cells; x++, i++)
    ...     {
    ...         TriangulateCell(
    ...             ● voxels[i],
    ...             voxels[i+1],
    ...             voxels[i+resolution],
    ...             voxels[i+resolution+1]);
    ...     }
    ...     if (x_neib != null)
    ...     {
    ...         ● triangulateGapCell(i);
    ...     }
    ... }
}
```

# Marching Squares, Voxel Class, Being dummy neighbor for voxels from other chunks to help bridging the gaps

Example of filling gap with x grid neighbor

Prepare for it

```
private void Triangulate()
{
    vertices.Clear();
    triangles.Clear();
    mesh.Clear();

    if (x_neib != null)
    {
        dummy_x.Become_X_DummyOf(x_neib.voxels[0], gridSize);
    }

    TriangulateCellRows();

    if (y_neib != null)
    {
        TriangulateGapRow();
    }

    mesh.vertices = vertices.ToArray();
    mesh.triangles = triangles.ToArray();
}
```



```
private void triangulateGapCell(int i)
{
    Voxel dummySwap = dummy_t;
    dummySwap.Become_X_DummyOf(x_neib.voxels[i + 1], gridSize);

    dummy_t = dummy_x;
    dummy_x = dummySwap;

    TriangulateCell(voxels[i], dummy_t, voxels[i + resolution], dummy_x);
}
```