

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.

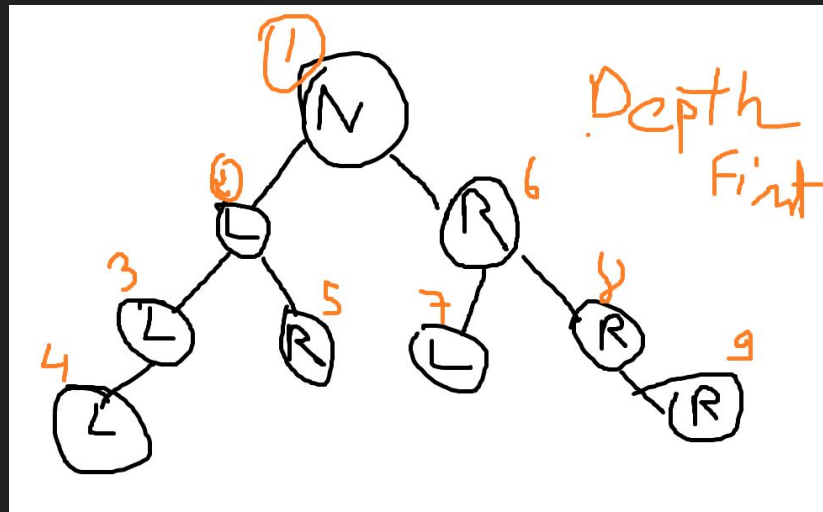
Implement as well a traverse breadth first and a travel depth first methods.

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>>();

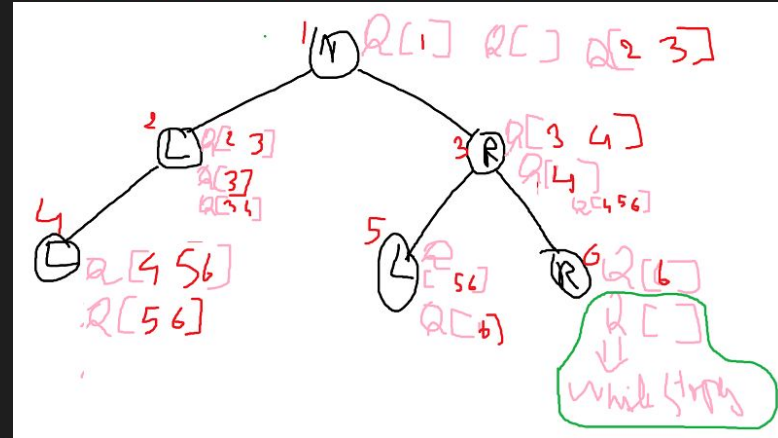
    queue.Enqueue(node);

    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);
        }
    }
}
```

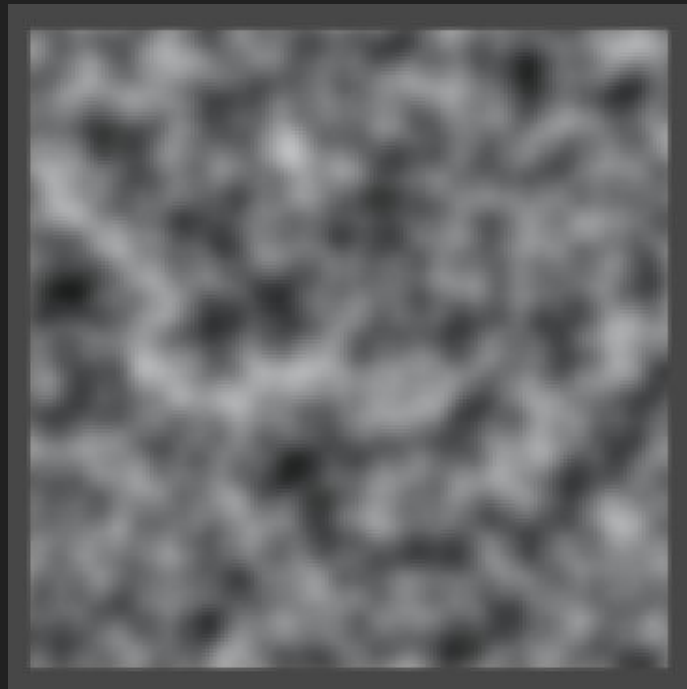


Perlin Noise

Perlin Noise

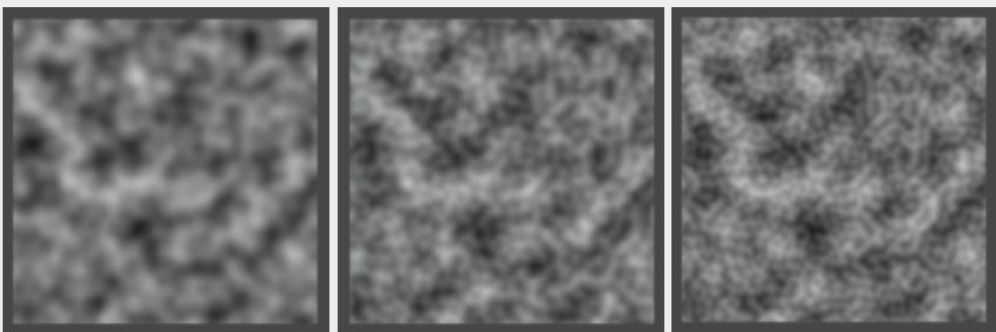
A [gradient fractal](#) noise, procedurally generates organic like textures.

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

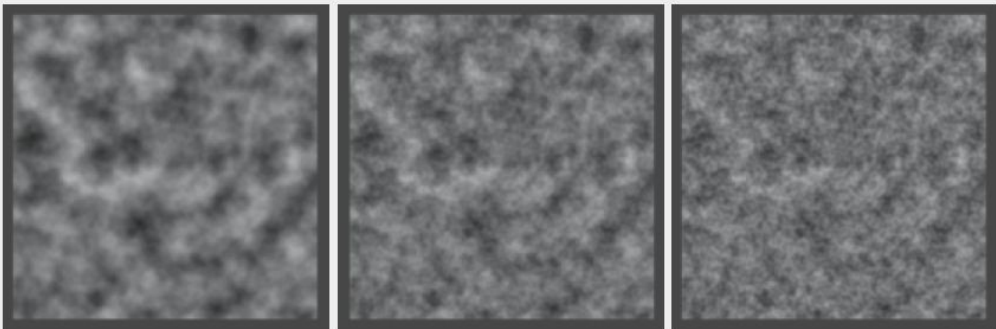


Perlin Noise

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;

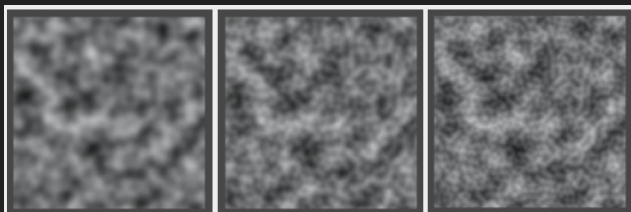
            float perlinValue = Mathf.PerlinNoise(
                sampleX, sampleY) * 2f - 1f;
            noiseHeight += perlinValue * amplitude;

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

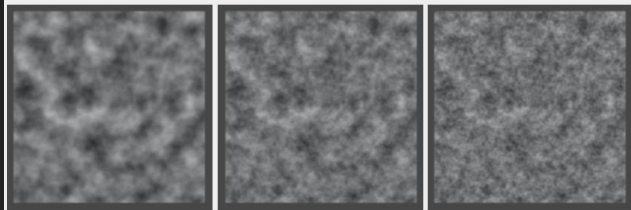
Perlin Noise: Assets\Lib\Scripts\Noise

- 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>();

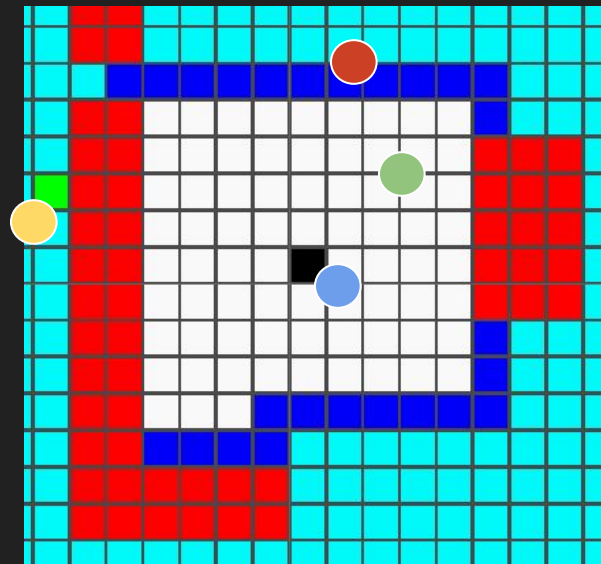
    frontier.Enqueue(start);
    reached.Add(start);

    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 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);

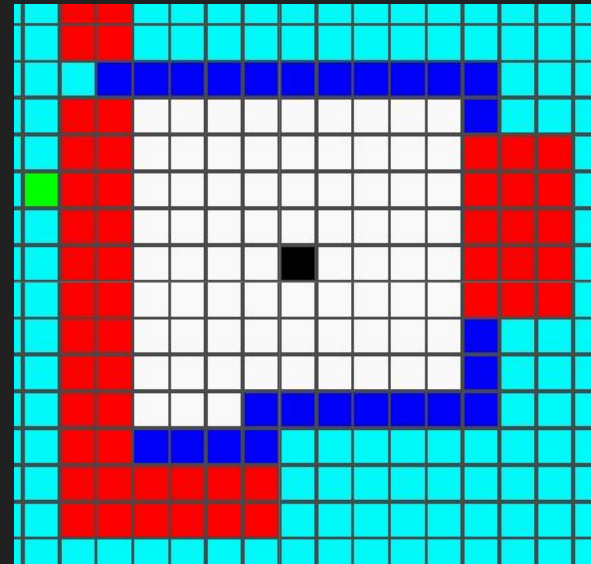
    while (frontier.Count > 0)
    {
        GridCell2D current = frontier.Dequeue();

        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 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);

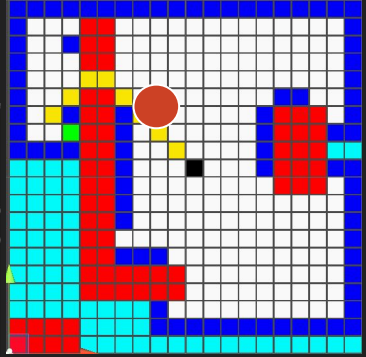
    while (frontier.Count > 0)
    {
        GridCell2D current = frontier.Dequeue();
        foreach (GridCell2D t in current.Neighbors)
        {
            if (t.IsWall())
                continue;

            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

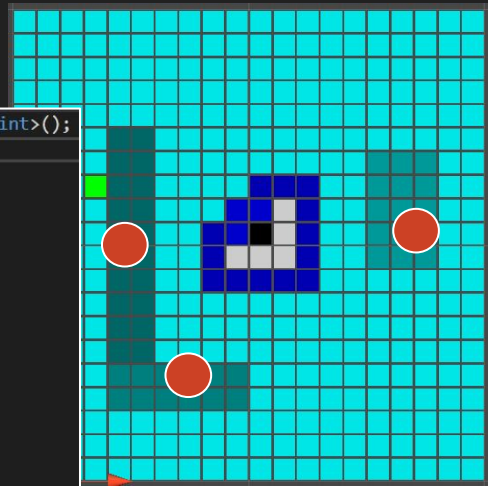
```
Dictionary<GridCell2D, int> costSoFar = new Dictionary<GridCell2D, int>();
costSoFar.Add(grid.StartCell, grid.StartCell.Cost);

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);

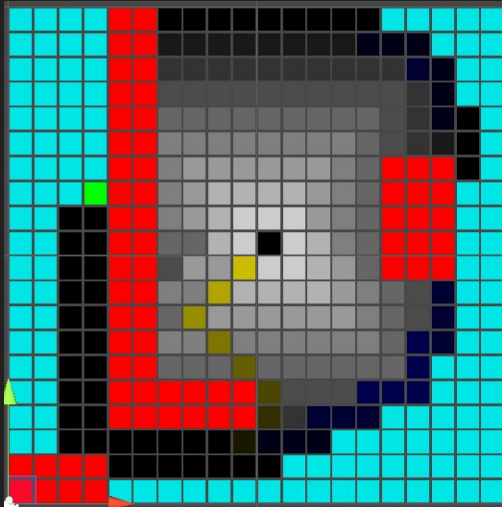
    foreach (GridCell2D next in current.Neighbors)
    {
        if (next.IsWall())
            continue;

        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;
            }

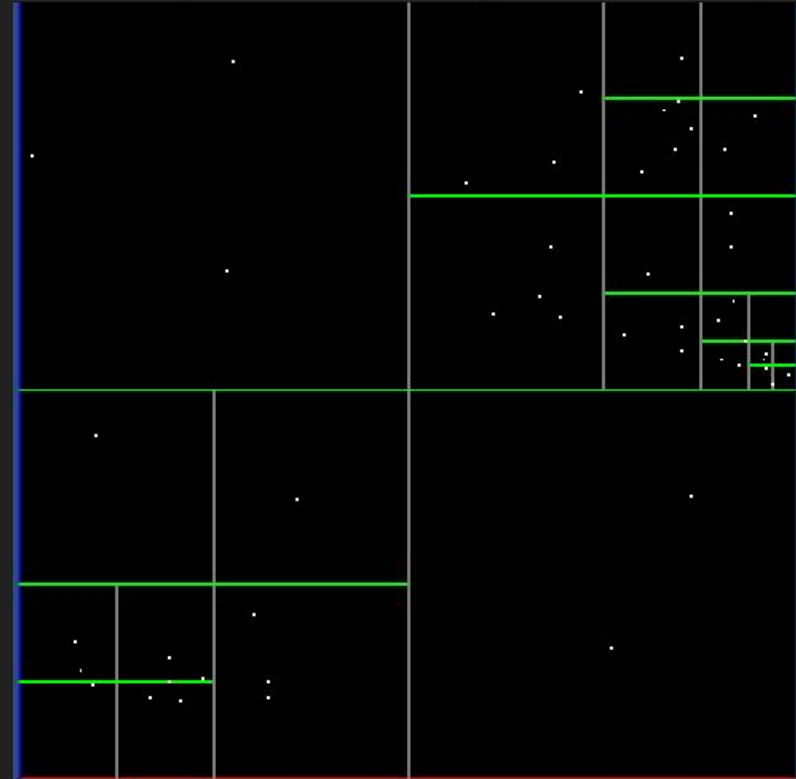
            if (next == grid.TargetCell)
            {
                next.DistanceToTarget = ManhattanDistance(next.Index, grid.TargetCell.Index);
                frontier.Add(next);
                frontier.Sort(new GridCell2DDistanceToTargetAndCostComparer());
            }
        }
    }
}
```

Quadtree

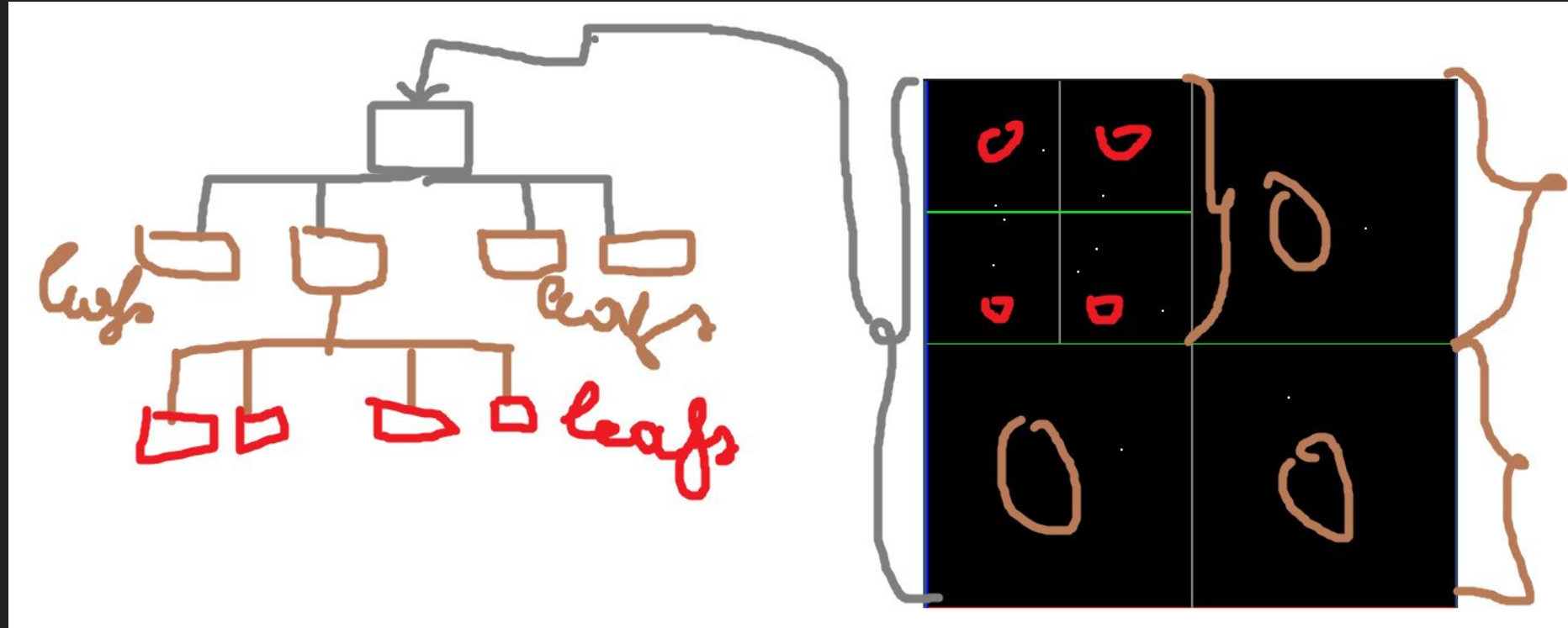
Quadtree

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

For example: more points leads to more partitions.

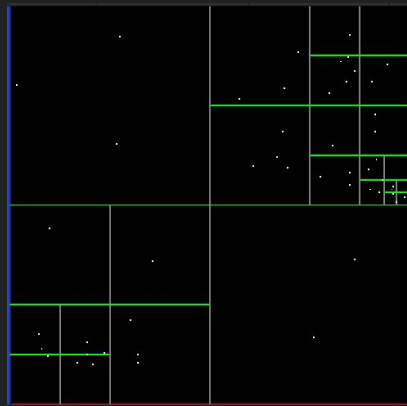


Quadtree, visualization



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;
    }
}
```

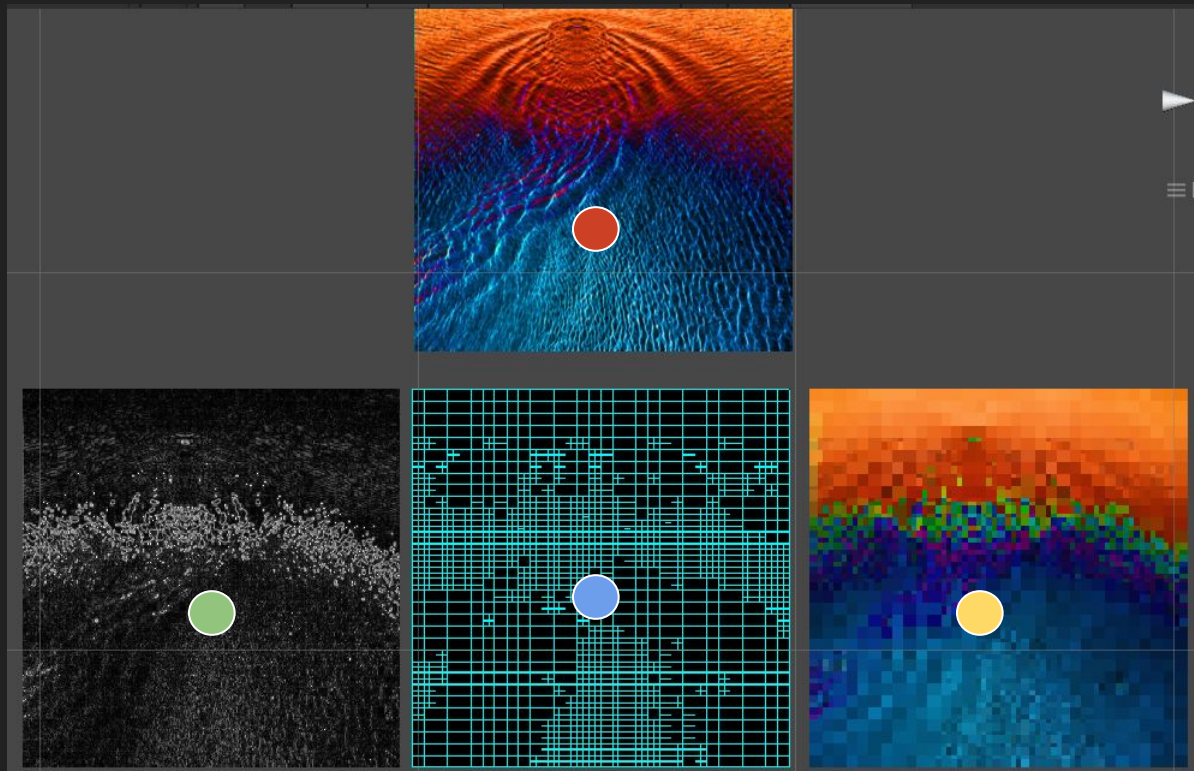
Quadtree, Image compression

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

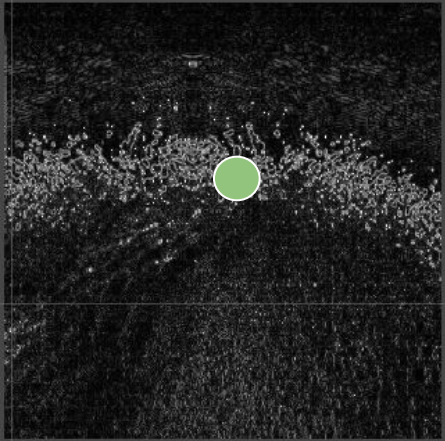
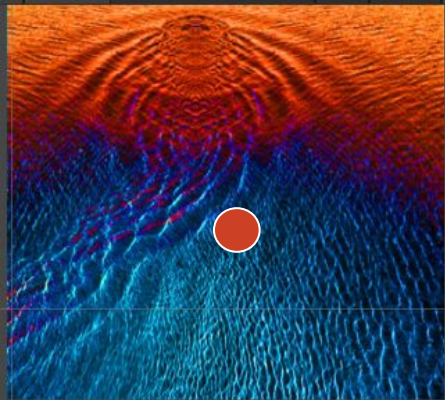
● Laplacian =

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

↑
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);
```

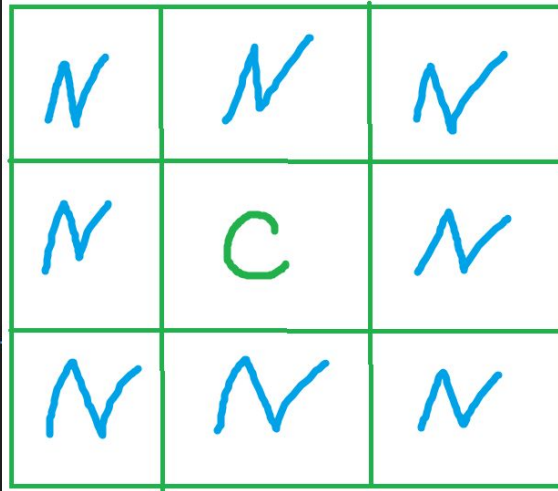
```
    neibs[7] = t.GetPixel(x, y - step);
```

```
    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;

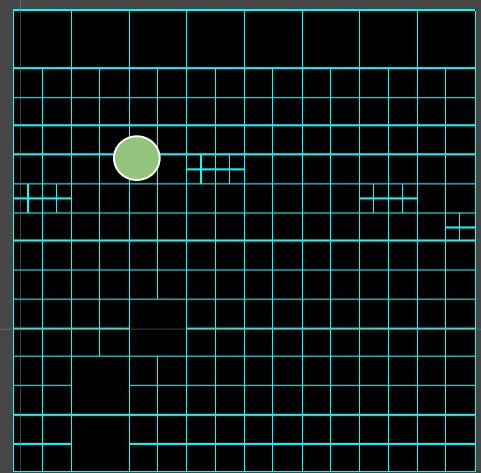
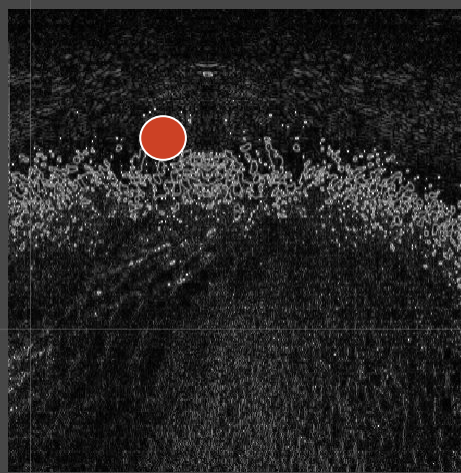
    DoSubdivide();
}

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

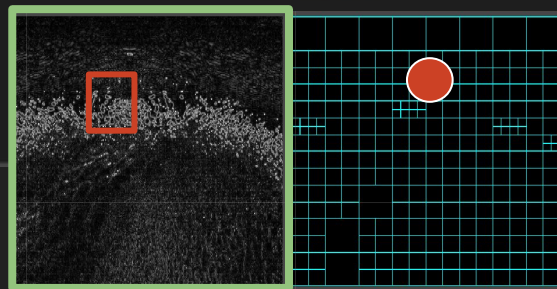
● Boundary quad
cumulating white pixels

● 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.

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

- Central point as dominant, or
- Average One

