MATHEMATICAL MODELING AND COMPUTING, Vol. 11, No. 1, pp. 1–18 (2024)

Using a compute shader for an adaptive particle system

Onufriienko D. M.

Lead Technical Artist, Pingle Studio, Dnipro, Ukraine

(Received 18 March 2023; Revised 2 September 2023; Accepted 9 September 2023)

The article proved the hypothesis of high efficiency of using a compute shader for a particle system being capable of tracking and adapting to other objects in a space of the game environment. A comparative description of the performance of the adaptive particle system based on CPU and GPU computing with additional optimization methods was given.

Keywords: *compute shader; GPU; particle system; rendering; Unity; VFX; video game.*

2010 MSC: 65K10 **DOI:** 10.23939/mmc2024.01.001

Abbreviations

GPU is a graphics processing unit;
CPU is a central processing unit
ASE is an amplify shader editor;
API is an application programming interface;
FPS is frames per second.

Nomenclature

B is a noise texture width;
H is a height of noise texture;
H' is a height of a player (a bot) tracking texture;
P is a coefficient of the protruding surface area under the player;
S is a coefficient of texture area in space;
t is a time;
v is a texture shift speed;
$x_{cp}(y_{cp}, z_{cp})$ is a position of the object in space for any axis in the current frame;
$x_{cp-1}(y_{cp-1}, z_{cp-1})$ is a position of the object in space for any axis in the previous frame;
$x_{pl}(y_{pl}, z_{pl})$ is a player (or a bot) position in space for any axis;
r is a rotation of particle;
$F()$ is a particle system structure;
$f()$ is a user criterion characterizing the argument quality relatively to the problem being solved;
"opt" is an optimal (desired or acceptable) value of the functional $f()$ for the problem being solved;
W is a set of transformation sets.

1. Introduction

Scientific research on the use of GPU-assisted math parsing is relevant in many areas of practical problems: from parsing sites to filtering noise affecting UAV imaging and control. This case will test the efficiency of calculating mathematical problems for a particle system on the GPU rather than on the CPU.

There is a similar commercially successful special effect Living Particles by SineVFX [1], where the system runs on the CPU. This system cannot be implemented on a "pure" ParticleSystem from Unity [2], because it is not possible to set the unique position of each particle and track the positions of bots.

There is created framework of the system without a compute shader in C#, and then it will be a system completed with optimization methods step by step. Each modification will be considered and recorded in the result table; depending on the project, the configuration of allowable methods may vary. These methods include rendering with Draw Mesh Instanced and Draw Mesh Instanced Indirect, writing data to vertexes, calculating positions on the CPU script or directly in the shader. If it is necessary to return the data from the GPU to the CPU, it is important to take into account the additional load that appears.

Using these shaders is preferable to writing your own data transport systems, as all additional functionality has already been created for the most popular video game development environments, such as Unreal Engine and Unity. It would also require a large financial and time investment.

The transfer of computations to the GPU may be required in many projects and games, as the same kind of computations would be performed hundreds of times faster [3,4]. Therefore, the research on computational shaders in the field of digital technology is relevant.

2. Problem statement

The system input parameters: B, H, H', P, S and v set by the user.

Position for particle without bots: $\{x_{cp}, y_{cp}, z_{cp}\}$ and with bots: $\{x_{pl}, y_{pl}, z_{pl}\}$. Then $W = [\{x_n, y_n, x_n, r\}, \{x_{n+1}, y_{n+1}, z_{n+1}, r\}, \ldots]$, and $\{x_n, y_n, z_n\} \in \{x_{cp} \vee x_{pl}, x_{cp} \vee x_{pl}, x_{cp} \vee x_{pl}\}$; $f(F(), W, \{x_n, y_n, z_n\}) > \text{opt}$, where n is particle index, model structure $F()$ is ported from the CPU to the GPU, and W is the output set of the function.

In turn, the problem is to determine the use of $x_{cp} \vee x_{pl}$ for a particular particle and how to calculate them.

The optimality criterion is the frame rate per second, the number of particles, and the number of tracked objects. The higher these indicators, the more efficient $F() > \text{opt}$.

3. Review of the literature

Understanding the basic optimization principles is described in the book [5], which will help in creating and upgrading any self-written particle systems.

In the research [6], there was a comparison of the following methods: particle systems based on three-dimensional vector field texture on the GPU and based on equations. However, as noted by the authors themselves, the first of the methods was effective only in specific situations and cannot replace the standard method of equations; the second method is slower.

Techland Co. Ltd. presented a GPU-based particle system creation and control system in 2016 [7]. The article [8] discusses fluid modeling using multithreading. However, the functionality of the above methods is not enough to implement an adaptive particle system.

The research [9] has shown that using geometry shader is not always preferable to using standard CPU methods. As compute shaders also run on the GPU, it is necessary to take into account the possible disadvantages of the chosen method.

However, not every task can be optimized by transferring calculations to the GPU, but only those that can be parsed such as collision of multiple spheres, real-time simulation of fabric [10], footprints on snow [11] or terraforming [12]. However, unique tasks performed once and not applied to multiple objects or data, will not be effectively computed on the GPU, e.g., player behavior.

The correct use of threads in the compute shader kernels constitutes a weighty part of the system performance. For more details on the principles of kernels, see the article [13]. In this case, 1024 threads will be set on one axis. This has shown the highest performance gain, however, each case is unique and requires an individual approach. In addition, knowing the structure of the computer components will help us in optimizing the compute shader.

In the researches [6–8] discussed above and others similar presented in contemporary sources of scientific periodicals, the question of the efficiency of using compute shaders for an adaptive particle system capable of the complex behavior of its particles remains untouched. Moreover, there is no comparative characteristics of the use of such a particle system with additional game optimization methods.

4. Materials and methods

Let us transfer the basic mathematical calculations of the position of a particle in space from the CPU to the GPU. As they make up the bulk of the system load, it will increase its performance.

In the experiment, the CPU is used to form matrices and buffers for object transformations. The task of the GPU consists of: drawing particles (HLSL shader), calculating the position of the particle (the first kernel of the compute shader) and forming a complex shape of the particle field (the second kernel).

The calculated object transformation matrices are immediately sent to the object shader, where their transform is formed. This allows you not to waste time returning data to the CPU.

Optimization methods were used writing data to the vertexes of the 3D model, computing particle positions in the HLSL shader, drawing using Draw Mesh Instanced and Draw Mesh Instanced Indirect. A comparative description of the use of these methods is also given. This will allow the developer, based on the information obtained, to choose the necessary parameters of the system for his own project.

5. Experiments

List of hardware and software used to measure system performance:

Component	**Specification**
OS	Windows 10 (USA)
CPU	Intel®Core^{TM} i7-5500U (USA)
GPU	NVIDIA GeForce GTX 1060 (USA)
RAM	8 GB
IDE	Unity 2018.4.20f1 (USA), Microsoft Visual Studio Community 2019 (USA)

Plan:

1) Create particle system on the CPU;
2) Calculate the particle positions in HLSL shader;
3) Offload the GPU using Draw Mesh Instanced;
4) Offload the CPU using the computational shader;
5) Replace Draw Mesh Instanced with Draw Mesh Instanced Indirect to compare them.

System input parameters: B, H, H', P, S and v are set by the user based on the visual representation of the system.

5.1. The basis of the system

This will use C# to mathematically calculate the motion of objects, and Amplify Shader Editor to visually represent how the system works in the first phase. Later, the shader made with ASE will be replaced by a self-written one in ShaderLab to be used along with the compute shader. A method will also be implemented that is capable of shaping complex particle field shape.

1. Creating the field with a loop in the Start function caused when launching the editor:

Algorithm 1 Start field.

```
1: Given: $x, z = 0$ is position in space;
2: Require: $W = \varnothing$ is array of column transforms;
3: for $i = 1, \dots,$ lengh of field
4: for $y = 1, \dots,$ width of field
5: $x := x +$ offset between columns;
6: instantiate column;
7: set position and rotation;
8: $W := W +$ column;
9: $z := z +$ offset between columns;
10: if $i\%2 = 0$
11: $x := 0$
```

2. Field landscape.

The field consists of two main textures. Noise textures are used to make the particles move chaotically [15] and target textures responsible for the shape of the particle landscape under the player and bots may look like this (Figure 1):

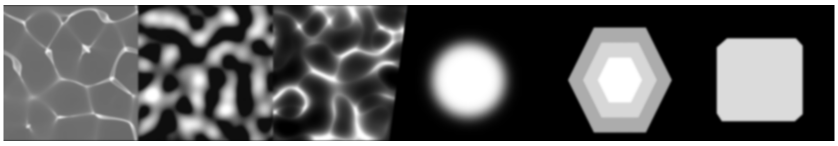


Fig. 1. Particle landscape textures.

Data reading and writing should be enabled in the noise texture settings. They should also be seamless textures. Raven and Perlin noises and their analogs would work well.

3. Formulas for calculating the column position relative to the texture pixel (1):

$$x_{cp} = \frac{x_{cp-1} - B}{S} + tv. \tag{1}$$

The formula is used for the X and Z axes with the width and length of the texture respectively. v is used to control animation speed.

Now the formula for the position of the column under the tracked object (2):

$$x_{cp} = \frac{x_{cp-1} - x_{pl}}{PB + B/2}. \tag{2}$$

The texture length and width are divided in half so that the texture responsible for the position of the columns under the object is in the center of the object according to the world coordinates rather than shifting to the right and up. The X and Y axes (where Y is up) are identical.

It is necessary to take a pixel from the first channel (Red), or the averaged RGB pixel-grid, where black is 0 meters, white is 1 meter on the Y axis.

However, at positions where a noise texture pixel is $\gtrsim 0.8$ and an object texture is $\lesssim 0.3$, the column goes strongly upwards. Because of equal texture priorities after remultiplying, we get too much value. We prioritize tracking texture over noise texture using multipliers. The formula (3) is:

$$x_{cp} = (1 - H')H + H'^2. \tag{3}$$

It is used interpolation for smooth motion of particles.

Also the values of the particle positions under the object can have both positive and negative values (Figure 2), which will make both convex and concave areas.

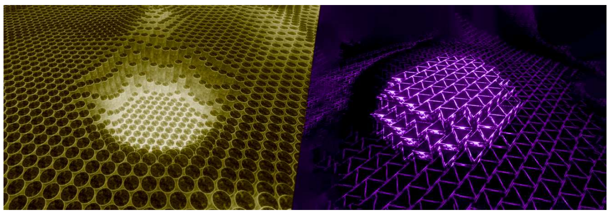


Fig. 2. Multiplier of the particle position under the player with negative and positive value.

At this point, it is necessary to transfer the data from the code to the material. Using a global variable or passing data directly into each material through GetComponent() proved more resource-intensive than writing it into mesh vertexes. However, the use of vertex data adds another calculating cycle and also the performance of this operation is directly related to the number of vertexes (a cubic column will be calculated faster than a regular hexagonal prism).

Algorithm 2 Set vertex color.

```
1: Given: $V$ is array of vertices;
2: Require: $C$ is array of colors;
3: $V$ := mesh.vertices;
4: for $i = 1, \ldots, V$ length
5: $C[i]$ := interpolation from $C[i]$ to new color;
6: mesh.colors := $C$;
```

```
void ColorMesh(Color newColor, Mesh mesh){
    vertices = mesh.vertices;
    colors = new Color[vertices.Length];
    for (int i = 0; i < vertices.Length; i++)
      colors[i] = Color.Lerp(colors[i], newColor, 1f);
    mesh.colors = colors;
}
```

4. Creating a 3D model, UV and glow mask.

Any 3D object program (Autodesk 3ds Max) will work for the mesh creation and UV particle unfolding.

Black and white texture to create the glow mask either manually (use Photoshop, Substance Painter or etc). The created texture is used in the shader for lighting the object, depending on its position on the Y-axis in the world.

5. Creating a shader using the ASE (Figure 3).

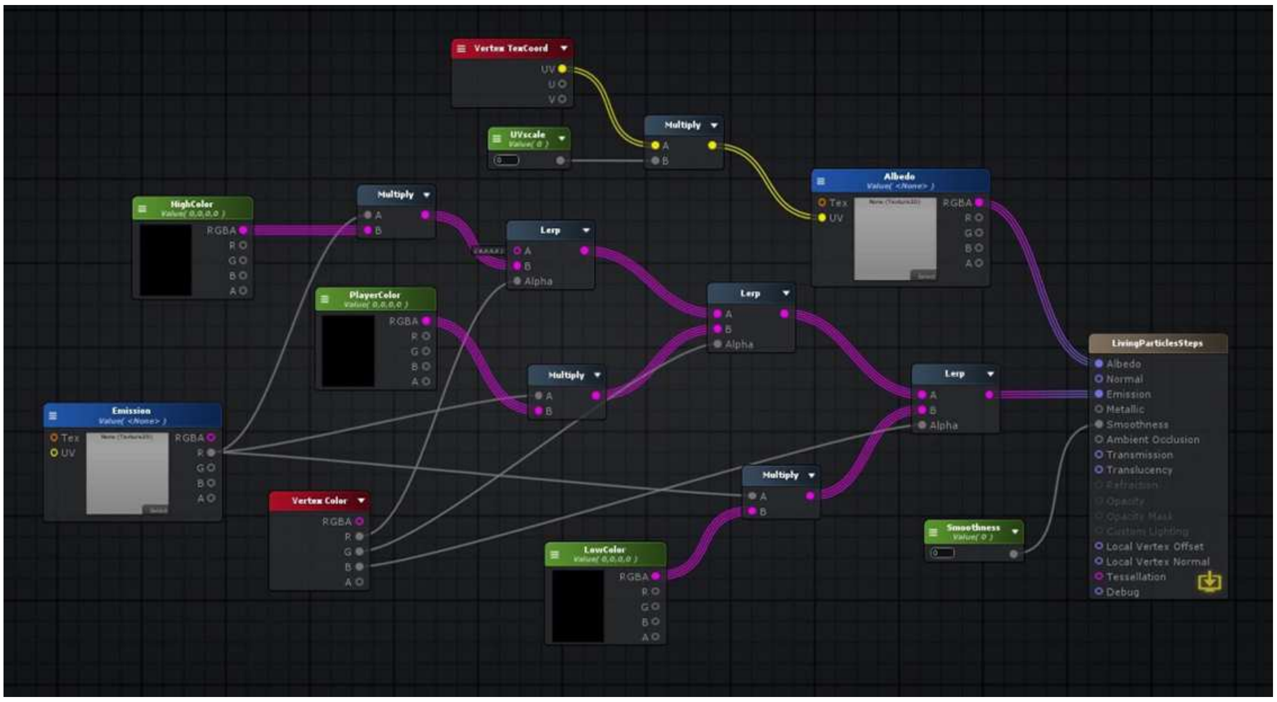


Fig. 3. The basis of the shader in the Amplify Shader Editor.

The Vertex Color node takes the recorded data in the code and passes the values for interpolation channel by channel. Each channel takes a texture multiplied by a layer's color.

6. Changing the shape of the field.

A project may not require a solid square field, but some kind of complex particle arrangement like a plan (Figure 4).

To set the gaps, the easiest way is to use an additional texture as a mask. Black represents the absence of a particle, white represents its presence. Determining whether a particle is present or absent is done using the formula: divide the number of particles by the length of the texture and add this number to the variable for each pass of the loop, respectively for the width.

Problem: filling the array will leave empty cells that will be sent to the video card for rendering. Consequently, whether there are 1000 particles or 1 particle and 999 passes, the FPS will be the same. It will not be possible to clean up the array on the CPU because the cycle that checks for empty cells and then shifts data will take 250 000 particles about 10 minutes.

To speed up the program, calculate the presence of particles in the compute shader. The kernel algorithm will look like this:

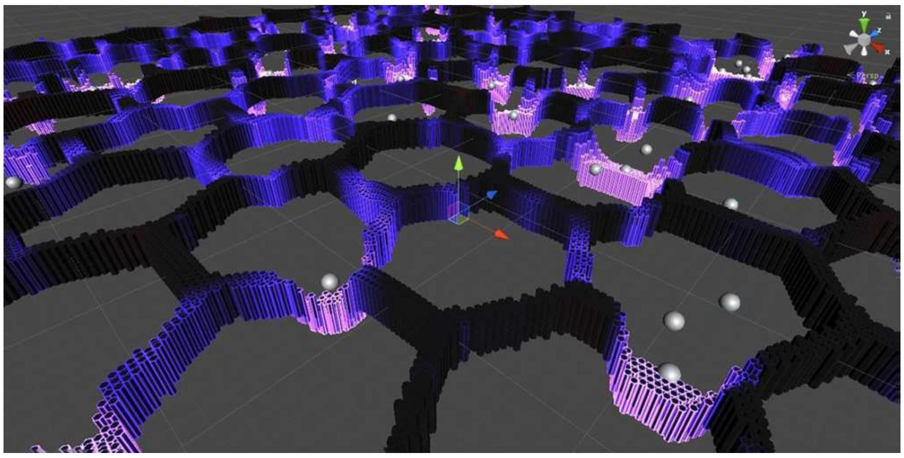


Fig. 4. A system using a particle arrangement mask.

Algorithm 3 GPU complex field.

```
 1: Given: $w$ is width of field; $l$ is length of field; $s$ is step; $x$, $z$ is offset of position; $sX$, $sZ$ are count steps
    along the axes; $id$ is dispatch thread ID (column index);
 2: Require: $b$ is buffer of columns;
 3: $s :=$ textureWidth$/w$;
 4: while $id > w$
 5:     $id := id - w$;
 6:     $x := x +$ offset;
 7:     $sX := sX + s$;
 8: while $0 < id$
 9:     $id := id - 1$;
10:     $z := z +$ offset;
11:     $sZ := sZ + s$;
12: if (maskTexture $\neq 0$) then
13:     $b[id] := (x, 0, z)$;
14:     else
15:     $b[id] := -1$; // show that this position is empty
```

```
#pragma kernel CreateField [numthreads(1024,1,1)]
void CreateField (uint3 id : SV_DispatchThreadID){
  CFstep = (float(columnsFieldWidth)) / (float(width));
  countColumnFieldPosition = id.x;
  while(countColumnFieldPosition > width){
     countColumnFieldPosition = countColumnFieldPosition - width;
     fieldX = fieldX + distanceX;
     CFx += CFstep;
  }
  while(0 < countColumnFieldPosition){
     countColumnFieldPosition = countColumnFieldPosition - 1;
     fieldZ = fieldZ + distanceZ;
     CFz += CFstep;
     if (offset == 0) offset = 1;
     else offset = 0;
  }
  // additional offset
  if(offset == 1 && useDistanceBias == 1)
     fieldX = fieldX + distanceX / 2;
  if(columnsField[float2(CFx, CFz)].x != 0){
     bufferForColumnArray[id.x].x = fieldX;
     bufferForColumnArray[id.x].y = 0;
     bufferForColumnArray[id.x].z = fieldZ;
  } else {
     bufferForColumnArray[id.x].x = -1;
  }
}
```

On the CPU side, we first take the data in List, set the length and put it into an array with a fixed length:

Algorithm 4 GPU complex field.

```
1: Given: $C$ is array of columns; $lC$ is list of columns; $nC$ is new array of columns (mask);
2: for $i = 1, \dots, C$.length
3:     if $C \neq -1$ then
4:         $lC := lC + C[1]$;
5: $nC :=$ new array with $lC$.length;
6: for $i = 1, \dots, lC$.length
7:     $nC := nC + lC$;
```

Since the position of each particle is determined in parallel, the problem will be solved ten times faster than any solution on the CPU.

7. Shader in ShaderLab.

On the ShaderLab we create a shader to render the object because the shader created in ASE v 1.6.4 cannot be used together with the compute shader:

1. Get the data from the positionBuffer and enter it into the matrix of the position of the particle in space. Similar to Draw Mesh Instance:

Algorithm 5 HLSL object transform.

```
1: Given: $lM$ is matrix of local transform; $wM$ is matrix of world transform; $B$ is position buffer;
2: if $B.w = 0$ then
       // check rotation
3:     $lM$:= {| 1    0    0    0 |
               | 0    1    0    0 |
               | 0    0    1    0 |
               |$B$.x $B$.y $B$.z  1 |};
4: else
5:     $lM$:= {| -1   0    0    0 |
               | 0    1    0    0 |
               | 0    0    -1   0 |
               |$B$.x $B$.y $B$.z  1 |};
6: $wM$[1,4][2,4][3,4][4,4] := |$B$.x $B$.y $B$.z 1 |;
7: $lM$[1,4][2,4][3,4] := $lM$[1,4][2,4][3,4]*-1;
8: $lM$[1,1][2,2][3,3] := 1/$lM$[1,1][2,2][3,3];
```

```
#ifdef UNITY_PROCEDURAL_INSTANCING_ENABLED
   data = positionBuffer[unity_InstanceID];
   if (data.w == 0) {
       unity_ObjectToWorld._11_21_31_41 = float4(1, 0, 0, 0);
       unity_ObjectToWorld._12_22_32_42 = float4(0, 1, 0, 0);
       unity_ObjectToWorld._13_23_33_43 = float4(0, 0, 1, 0);}
   else{
       unity_ObjectToWorld._11_21_31_41 = float4(-1, 0, 0, 0);
       unity_ObjectToWorld._12_22_32_42 = float4(0, 1, 0, 0);
       unity_ObjectToWorld._13_23_33_43 = float4(0, 0, -1, 0);}
   unity_ObjectToWorld._14_24_34_44 = float4(data.xyz, 1);
   unity_WorldToObject = unity_ObjectToWorld;
   unity_WorldToObject._14_24_34 *= -1;
   unity_WorldToObject._11_22_33 = 1.0f / unity_WorldToObject._11_22_33;
#endif
```

2. The initial shader logic is either written by hand or copied from the original AMP shader. The algorithm is shown in Figure 3. HLSL code:

```
void surf (Input IN, inout SurfaceOutputStandard o) {
  float4_Emission_ST = float4(1,1,0,0);
  // first layer of calculations
  float2 uv_Emission = IN.uv_Emission * _Emission_ST.xy + _Emission_ST.zw;
  float4 tex2DNode = tex2D(_Emission, uv_Emission);
  float temp_output = ((data.y - startPositionY + _HighColorAdd) * _HighColorMultiply);
  // second layer of calculations
```

```
   float clampResult1 = clamp( temp_output , 0.0 , 1.0 );
   float4 lerpResult1 = lerp(float4( 0,0,0,0),(tex2DNode.r*_HighColor),clampResult1);
   float clampResult2 =
        clamp(((temp_output +_LowColorAdd)*_LowColorMultiply),0.0,1.0);
   // third layer of calculations
   float clampResult3 = clamp(((1.0 - clampResult1) * clampResult2) , 0.0 , 1.0);
   float4 lerpResult2 = lerp(lerpResult1,(_MiddleColor * tex2DNode6.r),clampResult3);
   float4 lerpResult3 = lerp((_LowColor * tex2DNode6.r ), lerpResult2, clampResult2);
   // set data
   o.Emission = lerpResult3.rgb;
 }
```

8. Getting performance results.

Problem: With the standard programming approach, the system faces a large load on the CPU and on the GPU. With 2000 particles and 1 tracked object, we have ~ 30 FPS (30 ms CPU, 29 ms GPU) and 3445 batts, which makes the system computationally unsuitable for use in a real project.

The CPU load is due to:

1) 60% of the performance is consumed by the Update function, where the main calculations take place. The function is called in every frame and has two cycles. The first cycle goes through all the particles to impose noise, the second cycle checks if the bots and the player have any effect on the movement of particles.
2) About 40% of the load is vertex data recording.
3) Also, a lot of bots affect CPU performance because the engine sends a command to the graphics API (e.g., OpenGL or Direct3D) to play the object on the screen.

The load on the GPU comes from:

Multiple Batch – FrameDebugger is crowded with playback calls. The large number of rendering calls has a negative effect on performance, because the object shader is considered separately for each particle.

With the system created on the CPU we get the initial data for comparison with the same system but implemented on the GPU (point 5.5).

5.2. Calculating the particle position in the "standard" shader

Eliminating data recording in vertexes will relieve the CPU from unnecessary cycles. It is necessary to determine the position of a particle by its Y coordinate. To do that, the Transform node is used (Figure 5):

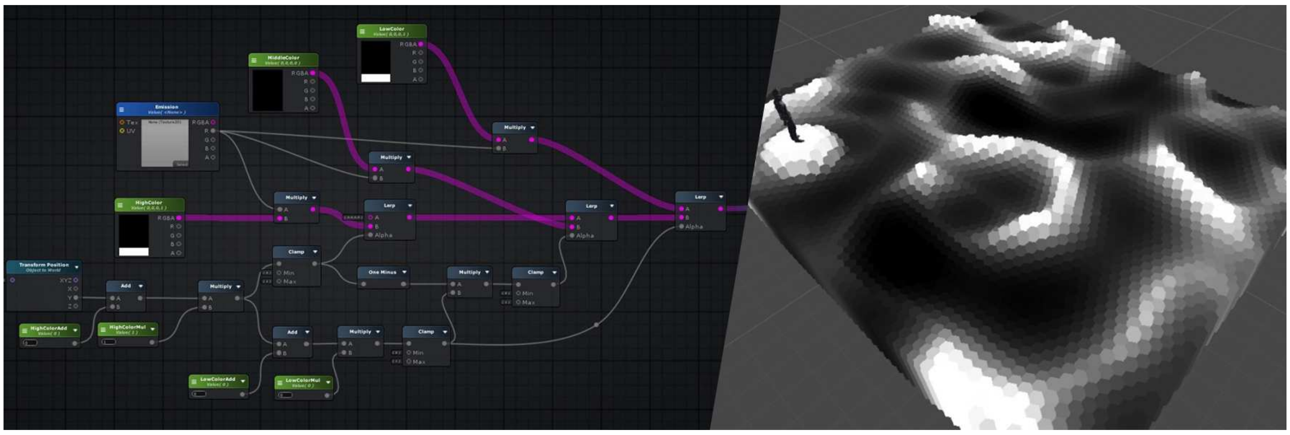


Fig. 5. Shader in the Amplify Shader Editor with the Transform node.

However, the node will work correctly with deferred lighting only, for forward lighting the shader should be switched off batching, which will also have a negative effect on performance [16].

Although for deferred rendering the number of rendering calls was reduced to less than a dozen, for forward rendering only the cycle of recording data in vertexes was reduced, the number of batches remains the same.

The result: a performance gain of $\sim 150\%$, 2000 particles and 1 object — 75 FPS (12 ms CPU, 5 ms GPU). This was due to elimination of writing data to vertexes on the CPU.

Conclusions: These measures are not enough to run the system efficiently. The CPU is still more loaded than the GPU. Further load reduction with Draw Mesh Instanced will reduce the load on both the CPU and the graphics card.

5.3. Drawing particles with Draw Mesh Instanced

Draw Mesh Instanced draws the same mesh multiple times using the GPU instance creation. Like Graphics.DrawMesh, this feature draws meshes for a single frame without the extra cost of creating unnecessary game objects.

1. The transformation matrix of each mesh instance must be packed into an array of matrices. The number of instances to render may be specified, or the default is the length of the matrix array. The rest of the data for each instance, if required by the shader, must be provided by creating arrays in the MaterialPropertyBlock argument using SetFloatArray, SetVectorArray, and SetMatrixArray.

When a field is created, the positions of the particles are entered into the matrix:

Algorithm 6 Field matrix.

```
1: Given: $w$ is width of texture; $l$ is length of texture; $x, z$ is position in space; $m$ is spawn mask texture;
2: Require: $M$ is array of matrices;
3: for $i = 1, \dots, w$
4:   for $j = 1, \dots, l$
5:     $x$ := offset;
6:     $id := i * l + j$;
7:     if texture pixel in this postion $\neq 0$ then
8:       $M[id]$.SetTransforms(position($x$,0,$z$), rotation, scale);
9:       $z$ := offset;
```

```
for (int i = 0; i < width; ++i) {
 for (int j = 0; j < depth; ++j) {
    columnPos.x += 0.36f;
    idx = i * depth + j;
    matrices[idx] = Matrix4x4.identity;
    countSpawnMaskDepth += spawnMaskDepth;
    // pixel value condition for putting the value into the matrix
    if (spawnMask.GetPixel(countSpawnMaskWidth, countSpawnMaskDepth).grayscale!=0)
    matrices[idx].SetTRS(new Vector3(columnPos.x, 0, columnPos.z),
 Quaternion.Euler(-90, 90, 0),scale);
 }
 columnPos.z += 0.36f;
}
```

2. As it is possible to send a maximum of 1024 objects for rendering, the data is written to an additional matrix. The first one saves the value of all particles, the second one only sends the necessary part for rendering.

Algorithm 7 Additional matrices.

```
1: Given: $c$ is number of columns; $n$ is number of batches; $w$ is width of texture; $l$ is length of texture; $u = 0$
   is count passed columns; $m = 0$ is the length of the matrix for drawing, taking into account the remainder;
   $const = 1024$ is max number of elements in the matrix;
2: Require: $M$ is array of additional matrices;
3: $c := w * l$;
4: $n := c/const$;
5: for $i = 1, \dots, n$
6:   $m :=$ minimum($const, c - (const * i)$);
7:   // use function to calculate position of columns
8:   new $M$ with length $m$;
9:   for $y = 1, \dots, m$
10:    $M[y]$ := main array of all matrices[$y + u$];
11:  $u := u + const$;
```

```
  void Update(){
    total = width * depth;
    batches = Mathf.CeilToInt(total / BATCH_MAX);
    count = 0;
    for (int i = 0; i < batches; ++i){
        batchCount = Mathf.Min(BATCH_MAX, total - (BATCH_MAX * i));
        ColumnPosition();
        batchedMatrices = GetBatchedMatrices(count, batchCount);
        Graphics.DrawMeshInstanced(mMeshFilter.sharedMesh, 0, meshMaterial,
  batchedMatrices, batchCount);
        count += BATCH_MAX;
    }
  }

  private Matrix4x4[] GetBatchedMatrices(int offset, int batchCount){
     batchedMatrices = new Matrix4x4[batchCount];
     for (int i = 0; i < batchCount; ++i)
         batchedMatrices[i] = matrices[i + offset];
     return batchedMatrices;
 }
```

3. In the ColumnPosition() function it is made all calculations of the particle position along the Y axis. Matrix4x4 stores object position data in the fourth column. It is also not possible to send matrix meshes that do not correspond to Transform for drawing, so as not to use the heavy function matrices SetTRS() setting the value of rotation, position and size, the value of matrix element a44 is manually replaced with 1.

Algorithm 8 Particle position.

```
 1: Given: $x, z = 0$ is position of pixel; $w$ is width of texture; $l$ is length of texture; $p = 1$ is column move
    height multiplier; $s = 0$ is save old color for future frame; $id$ is dispatch thread ID;
 2: Require: $W = \varnothing$ is array of positions;
 3: for $i = 1, \ldots, w$
 4:    for $j = 1, \ldots, l$
 5:       $id := i * l + j$;
 6:       if $(p > 0)$ then
 7:          $s = 0$;
 8:       else
 9:          $s = 1$;
10:       if has targets then
11:          $x, z :=$ by formula 2;
12:          $s :=$ fieldTexture.GetPixel($x$, $z$);
13:          $W[id] :=$ add all data;
```

```
 private void ColumnPosition(){
     for (int i = 0; i < width; ++i){
       for (int j = 0; j < depth; ++j){
         idx = i * depth + j;

         if (PlayerHighMul > 0) saveOldColor = 0;
         else saveOldColor = 1;

         dontOverDraw = false;
         if (!matrices[idx].isIdentity){
            if (targets.Count == 0){
                Gx = Mathf.FloorToInt((matrices[idx].GetColumn(3).x/(width/2.8f)*
       groundHeightmap.width)+ Time.time * noiseMovingX);
                Gz = Mathf.FloorToInt((matrices[idx].GetColumn(3).z/(depth/2.8f)*
       groundHeightmap.height)+ Time.time * noiseMovingY);
                saveOldColor = groundHeightmap.GetPixel(Gx, Gz).grayscale;
            }

            matrices[idx].SetColumn(3, new Vector4(matrices[idx].GetColumn(3).x,
      Mathf.Lerp(matrices[idx].GetColumn(3).y, saveOldColor, 0.2f),
      matrices[idx].GetColumn(3).z, 1));
          }
       }
     }
  }
```

Result: 75 FPS (12 ms CPU, 0.6 ms GPU). Although the load on the GPU decreased by 90%, it had no effect on the total FPS, because FPS is calculated by the lower figure, and the CPU is still loaded.

Conclusion: Draw Mesh Instanced only affected the GPU calculations. The number of batches remained unchanged, because of the Transform node in the shader.

5.4. Creating a Compute Shader

The previous steps have greatly unloaded the GPU, but the CPU was still a bottleneck. At this stage, the GPU latency is only 0.6 ms, while the CPU latency is 12 ms, so the GPU is waiting for the CPU to do its work.

A big performance boost will come from the compute shader which transfers all mathematical computations to the video card, as the video card easily computes complex shaders which are themselves just mathematical functions. The data is sent to the GPU as if it were a regular shader, and the resulting computational results are not used for rendering, as usual, but for other purposes.

A compute shader was introduced by DirectX 11 to compute parallel tasks of the same type. It is an array of threads that is divided into groups and each group is split into threads. The threads have access to data arrays, structures, vertexes, etc.

The main script is upgraded first, later the compute shader is written.

On the CPU side:

1. Initializing the buffer and kernel in C# is as follows:

Algorithm 9 Initializing of buffer.

```
1: Given: $W \neq \varnothing$ is list of columns after using the mask;
2: Create a new buffer with long and list data $W$;
3: Sending buffer to the compute shader;
```

```
columnsNodesBuffer = new ComputeBuffer(columnsArrayAfterMask.Length, 16);
columnsNodesBuffer.SetData(columnsArrayAfterMask);
kiCalc = _shader.FindKernel("UpdateColumnsPosition");
_shader.SetBuffer(kiCalc, "columnsNodesBuffer", columnsNodesBuffer);
```

2. In Update we call the kernel, assigning threads:

Algorithm 10 Threads count.

```
1: Given: $w = 2048$ is width of texture; $l = 2048$ is length of texture; $c = 0$ is number of threads;
2: $c := w * l/1024$;
3: sending required number of threads;
```

```
countDispatch = Mathf.FloorToInt(width * depth / 1024);
_shader.Dispatch(kiCalc, countDispatch, 1, 1);
```

3. These positions of the particle must be set by a new class. The variables x, y, z save the positions of the object in world coordinates, respectively. The fourth variable w will rotate the particle by 180 degrees, but even if it is not used, the presence of the fourth variable will preserve the total multiplicity of 128.

4. All buffers must be cleared after the editor or game is turned off:

```
if (columnsNodesBuffer != null)
    columnsNodesBuffer.Release();
columnsNodesBuffer = null;
```

On the GPU side:

1) Kernel announcement:

```
#pragma kernel UpdateColumnsPosition [numthreads(1024,1,1)]
void UpdateColumnsPosition (uint3 id : SV_DispatchThreadID){
...
```

The kernel has an input parameter id that stores a three-dimensional stream index, in this case 1024 streams. The index for the buffers will be set to [id.x] and [id.xy] for the texture.

2) Kernel can accept buffers and textures for both reading and writing:

```
RWStructuredBuffer<cubePosition> cubesNodesBuffer;
StructuredBuffer<cubePosition> targetsPos;
RWTexture2D<float4> targetHeightmap; Texture2D<float4> textureMap;
```

3) Methf.Lerp does not exist in HLSL, formula (4) must be set manually:

$$x = x_1 + (x_2 - x_1)P, \tag{4}$$

where x_1 is old value of particle, x_2 is target value of particle position.

```
Buffer[id.x].y = Buffer[id.x].y + (newPosition.x - Buffer[id.x].y) * 0.3f;
```

4) Main algorithm:

Algorithm 11 Compute shader kernel.

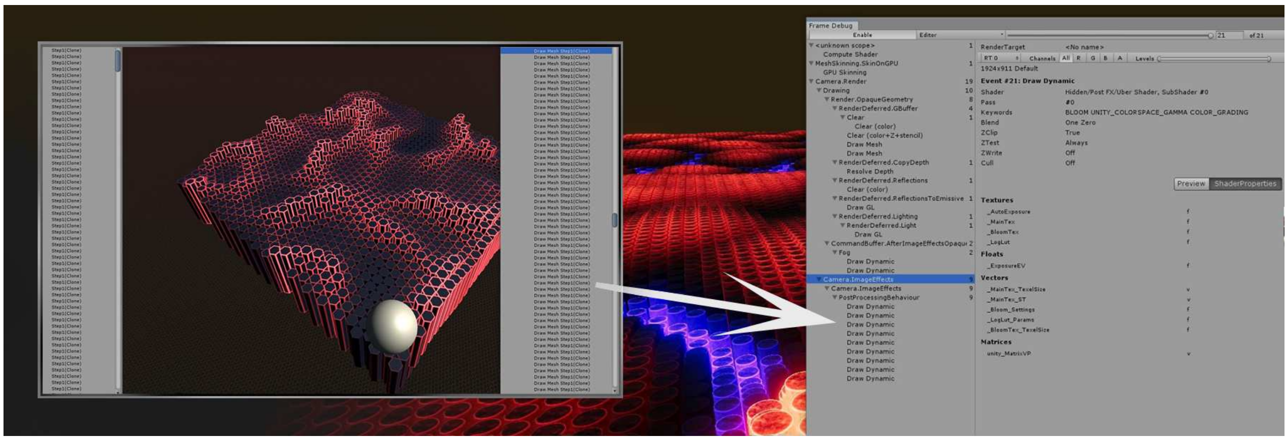
```
1: Given: $x, z = 0$ is position of pixel; $p = 1$ is column move height multiplier; $F = 0$ is field texture data by pixel; $B = 0$ is bot texture data by pixel; $s = 0$ is save old color for future frame; $m = 0$ is column movement height; $t = false$ is find target; $b$ is buffer; $id$ is dispatch thread ID;
2: Require: $W = \varnothing$ is array of positions;
3: if $(p >= 0)$ then
4:     $s = 0$;
5: else
6:     $s = 1$;
7: for $i = 1, \dots,$ numberOfTargets
8:     if $t$ then
9:         // bot texture position estimation
10:        $x, z :=$ by formula 1;
11:        $F :=$ fieldTexture.GetPixel$[x, z] *$ power;
12:        if $F < 1$ then
13:            $x, z :=$ by formula 2;
14:            while $x, z >$ texture width
15:                $x, z := x, z -$ texture width;
16:                $B :=$ botTexture.GetPixel$[x, z] *$ power;
17:                // use multipliers
18:                if $B >= 0$ then
19:                    $m :=$ by formula 3;
20:                else
21:                    $m :=$ by formula $3 * -1$;
22:        else
23:            if $(F >= 0)$ then
24:                $m := F^2$;
25:            else
26:                $m := -F^2$;
27:        if $(p > 0 \;\&\; m > s)$ then
28:            $s := m$;
29:        else
30:            if $(F \neq 0)$ then
31:                if $(m < s)$ then
32:                    $s = m$
33:                else
34:                    $s = m$
35:        if fieldTexture.GetPixel$[x, z] = 1$ then
36:            $t :=$ true;
37:            if $(p > 0)$ then
38:                $s := F^2$;
39:            else
40:                $s := -F^2$;
41:            // if no targets to control
42:            if numberOfTargets $= 0$ then
43:                $x, z :=$ by formula 2;
44:                while $x, z >$ texture width
45:                    $x, z := x, z -$ texture width;
46:                $s :=$ fieldTexture.GetPixel$[x, z]$;
47:            $b[id] := b[id] + (s - b[id]) *$ multiplier; // formula 4
48:            $W[id] :=$ set position; // calculate position
```

```
for (int i = 0; i < Columns.Count; i++){
  // bot texture position estimation
  botsPixelX=Mathf.FloorToInt((Columns[i].position.x-
target.position.x)/PlayerTargetScale*targetHeightmap.width+(targetHeightmap.width / 2));
  botsPixelZ=Mathf.FloorToInt((Columns[i].position.z-
target.position.z)/PlayerTargetScale*targetHeightmap.height+(targetHeightmap.height / 2));
  // getting texture pixel
  targetHight=targetHeightmap[float2(botsPixelX, botsPixelZ)].x*PlayerHighMul;
  // terrain calculations for a particle
  if (targetHeightmap[float2(botsPixelX, botsPixelZ)].x<1){
      landPixelX=(int((cubesNodesBuffer[id.x].x/scale*textureWidth)+time *noiseMovingX));
      landPixelZ=(int((cubesNodesBuffer[id.x].z/scale*textureHeight)+time *noiseMovingY));
      // offset for texture duplication
      while (landPixelX>textureWidth)
         landPixelX=landPixelX-textureWidth;
      while (landPixelZ>textureWidth)
         landPixelZ=landPixelZ-textureWidth;
      ...
```

The difference from calculation of texture data on the GPU will be that the texture will not be duplicated to infinity on the x- and z-axes, as it was on the CPU. The texture will behave similarly to Wrap mode - Clamp. To get the desired pixel, we artificially cut off the traversed parts of the texture (line 12 and 37).

Result: 25 FPS (35 ms CPU, 30 ms GPU) with 1,000,000 particles and 10 objects. Although all calculations were transferred to the GPU, the CPU remains busy because of the large number of objects filling the buffers. Changing from Draw Mesh Instanced to Draw Mesh Instanced Indirect will add performance, but will require code changes and a new shader.

As expected, all the particles are drawn in one call from the GPU side. There is comparison of old and new profiling (Figure 6):



```
Frame Debug
Enable    Editor                                   21   of 21
▼ <unknown scope>                          1   RenderTarget      <No name>
    Compute Shader                             RT 0   Channels All R G B A   Levels
▼ MeshSkinning.SkinOnGPU                   1   1924x911 Default
    GPU Skinning                               Event #21: Draw Dynamic
▼ Camera.Render                           19   Shader     Hidden/Post FX/Uber Shader, SubShader #0
  ▼ Drawing                               10   Pass       #0
    ▼ Render.OpaqueGeometry                8   Keywords   BLOOM UNITY_COLORSPACE_GAMMA COLOR_GRADING
      ▼ RenderDeferred.GBuffer             4   Blend      One Zero
        ▼ Clear                            1   ZClip      True
            Clear (color)                      ZTest      Always
          Clear (color+Z+stencil)              ZWrite     Off
          Draw Mesh                            Cull       Off
          Draw Mesh
      ▼ RenderDeferred.CopyDepth           1
          Resolve Depth
      ▼ RenderDeferred.Reflections         1                              Preview  ShaderProperties
          Clear (color)
      ▼ RenderDeferred.ReflectionsToEmissive 1  Textures
          Draw GL                                _AutoExposure                      f
      ▼ RenderDeferred.Lighting            1     _MainTex                           f
        ▼ RenderDeferred.Light             1     _BloomTex                          f
            Draw GL                              _LogLut                            f
    ▼ CommandBuffer.AfterImageEffectsOpaqu 2   Floats
      ▼ Fog                                2     _ExposureEV                        f
          Draw Dynamic                         Vectors
          Draw Dynamic                           _MainTex_TexelSize                 v
  ▼ Camera.ImageEffects                    9     _MainTex_ST                        v
    ▼ Camera.ImageEffects                  9     _Bloom_Settings                    f
      ▼ PostProcessingBehaviour            9     _LogLut_Params                     f
          Draw Dynamic                           _BloomTex_TexelSize                f
          Draw Dynamic                         Matrices
          Draw Dynamic                           unity_MatrixVP                     v
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
```

Fig. 6. Profiling the system using a compute shader.

Thus, the GPU defines a set of particles as one mesh.

5.5. Optimization with Draw Mesh Instanced Indirect

Similar to Draw Mesh Instanced, this function draws many instances of the same mesh, but unlike the first method, its arguments specify how many instances will be drawn.

The buffer with arguments bufferWithArgs should have five integers with the given offset argsOffset: number of indexes for an instance, number of instances, location of source index, location of base vertex, location of source instance.

Draw Mesh Instanced Indirect is more convenient to use with a compute shader than Draw Mesh Instanced, because the work will be done only with buffers and it is not necessary to translate data from matrices.

Draw Mesh Instanced Indirect is embedded in the CPU code:

1. Buffer announcement:

```
void Start() {
  argsBuffer = new ComputeBuffer(1,args.Length*sizeof(uint),ComputeBufferType.IndirectArguments);
  UpdateBuffersForDrawShader();
  ...
```

2. Function for filling arguments and position buffer:

Algorithm 12 Draw mesh instanced indirect arguments.

```
1: Given: $bP$ is position buffer; $aP$ is position array; $C$ is columns field;
2: ensure submesh index is in range;
3: clear $bP$ and $aP$;
4: for $i = 1, \ldots,$ number of instances
5:     $aP[i]. := |C[i].x, C[i]..y, C[i]..z, 1|$;
6:     send buffers to column material;
7:     calculate indirect arguments for drawing;
8:     send $bP$ to compute shader;
```

```
if (instanceMesh != null)
    subMeshIndex = Mathf.Clamp(subMeshIndex, 0, instanceMesh.subMeshCount - 1);
if (positionBuffer != null)  positionBuffer.Release();

positionBuffer = new ComputeBuffer(instanceCount, 16);
positions = new Vector4[instanceCount];
for (int i = 0; i < instanceCount; i++)
    positions[i] = new Vector4(colArrMask[i].x, colArrMask[i].y, colArrMask[i].z, 1);

positionBuffer.SetData(positions);
instanceMaterial.SetBuffer("positionBuffer", positionBuffer);
instanceMaterial.SetFloat("startPositionY", startPosition.y);

if (instanceMesh != null){   // Indirect args
   args[0] = (uint)instanceMesh.GetIndexCount(subMeshIndex);
   args[1] = (uint)instanceCount;
   args[2] = (uint)instanceMesh.GetIndexStart(subMeshIndex);
   args[3] = (uint)instanceMesh.GetBaseVertex(subMeshIndex);}
else{
   args[0] = args[1] = args[2] = args[3] = 0;}

argsBuffer.SetData(args);
_shader.SetBuffer(kiCalc, "positions", positionBuffer);
```

3. Drawing in update function:

```
Graphics.DrawMeshInstancedIndirect(instanceMesh, subMeshIndex, instanceMaterial,
newBounds(Vector3.zero, new Vector3(1000.0f, 1000.0f, 1000.0f)), argsBuffer);
```

4. Also, at the end of the program it is necessary to clear the buffers:

```
if (positionBuffer != null)
    positionBuffer.Release();
positionBuffer = null;
```

Result: 30 FPS (30 ms CPU, 25 ms GPU) with the same parameters, which gives 20% performance from the previous measurement.

6. Results

The result of the overall performance increase is 1 000 000 particles and 10 bots (equivalent to 250 000 particles and 100 bots).

Figure 7 shows that the particles of the system form the landscape and correctly respond to the entry of bots into the area of action.

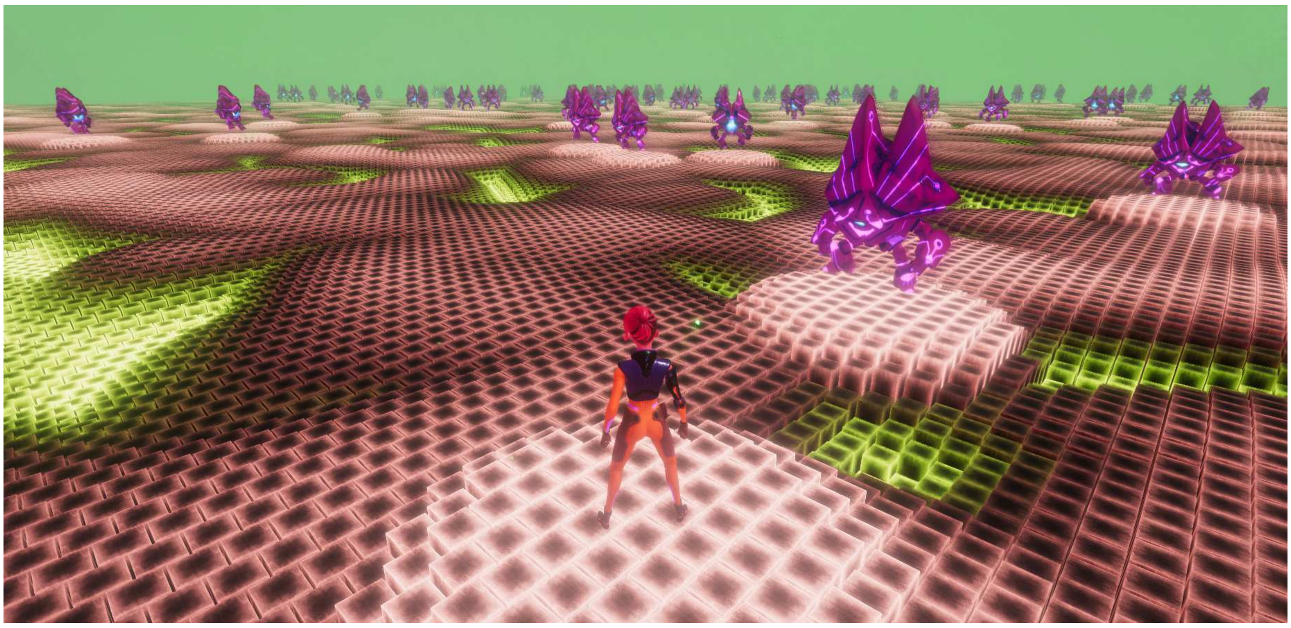


Fig. 7. Final result.

Table 1. Performance gain comparison.

Method	Number of columns/objects	Delay CPU, GPU (ms)	FPS
Percentage productivity increase over the previous method			
Basic method	2 000 / 1	30 / 30	30
Transform	2 000 / 1	12 (60%) / 5 (500%)	75 (150%)
DrawMeshInstanced	2 000 / 1	12 / 0.6 (733%)	75
ComputeShader	1 000 000 (50 000%) / 10 (900%)	35 (–65%) / 30 (–4900%)	25 (–66 %)
DrawMeshInstancedIndirect	1 000 000 / 10	30 (14%) / 25 (16%)	30 (20%)
Increase in productivity in percent relative to the base method			
Basic method	2 000 / 1	30 / 30	30
Transform	2 000 / 1	12 (60%) / 5 (500%)	75 (150%)
DrawMeshInstanced	2 000 / 1	12 (60%) / 0.6 (4900%)	75 (150%)
ComputeShader	1 000 000 (50 000%) / 10 (900%)	35 (–14%) / 30	25 (–16%)
DrawMeshInstancedIndirect	1 000 000 (50 000%) / 10 (900%)	30 / 25 (20%)	30

Let us add all mentioned modifications to Table 1 step by step and calculate the increase of productivity in percent thanks to their usage:

The percentage indicates the increase or decrease in performance in relation to the base method on the GPU (1st part of the table) or compared to the previous method (2nd part of the table).

Table 1 also shows that each additional optimization method brought the system closer to optimality. But moving the math to the GPU had the biggest effect on $F() \to \text{opt}$.

The calculations of the logic of the behavior of bots and the player are included in the measurements.

7. Discussion

The main problem of the particle system on the CPU is the inability to compute a large number of objects (Table 1). However, as expected, the transfer of calculations from the CPU to the GPU made it possible to increase the number of particles by hundreds of times (Figure 7). This is due to the fact that the GPU has great possibilities for paralleling calculations, and each particle is able to act as a separate mathematical representation in such a system. The second problem was a large number of rendering calls, the number of which has also been reduced (Figure 6). And the use of methods that can accelerate the calculation of computational tasks is the basis for the creation and porting of games to all kinds of platforms, where the main indicator of the success of the game optimization is the number of FPS.

The use of the compute shader has not yet become widespread, primarily due to the specifics of their creation and integration into projects. An attempt of global transfer of computational data to the GPU was proposed by the developers of Ultimate Battle Simulator 2 [17], where it was applied to AI bots and physics, which allowed using millions of units.

As compute shaders have a broader representation in popular cross-platform game development environments for computer games, there is much more functionality to reproduce the results in your own project. Also, as Unity can create builds for almost any platform without any extra operations, it is the use of a compute shader is a priority compared to Cuda or OpenCl.

Using the computational power of the GPU was an order of magnitude more effective than using the CPU for a special particle system. The hypothesis was tested on a system capable of calculating the position of a million particles, depending on the position of bots and the required landscape. A comparative characterization of additional optimization modifications in synthesis with a compute shader was performed. This complements the question of the relevance of using GPUs to solve such problems [9]. The methods presented by other authors [6–8] are good for creating standard effects (e.g., fire, smoke), but not for such adaptive particle systems.

The limitation of using such a method may be the initial load of the GPU. In this case, it is worth using a number of optimization methods described in the conclusions of the article, which produce more operations on the CPU.

The disadvantage remains the fact that with the functionality of this system, it is impossible to implement effects of another kind. For example, the capabilities of the standard particle system Unity or Unreal Engine. However, on the basis of this system, you can create an identical solution yourself.

Every system is not perfect, so you can always find better ways to implement the task at hand. Perhaps, this article is a comparative characterization of not all optimization capabilities which leaves a field of new opportunities.

Theoretically, a possible effective addition to this system would be to create an Occlusion Culling, so that the particles that are not in the frame are not processed. Under such conditions, it would be possible to specify an imperceptible bias in the particle system to make the field without boundaries itself. Such a system could be used, for example, as a place where the player is waiting for the main location to load.

8. Conclusions

1. A CPU-based particle system is computationally much slower than a GPU-based one. The easiest way to shape the field is to use a mask texture and clean up unused particles in a separate compute shader. The shader written in ShaderLab covers all necessary particle rendering needs in combination with a compute shader.

2. Node Transform took most of the load off the CPU and the GPU. With the same number of objects the increase in FPS was about 150%. This already makes the system usable for a project if bots and a large number of particles are not necessary.

3. Adding Draw Mesh Instanced did not add performance on the CPU, but the GPU got even greater gains. This method can be an alternative to the previous method if the latter is not possible in the use of the project.

4. Transferring the mathematical functionality to the compute shader has achieved the main goal — the number of particles is now measured in hundreds of thousands. However, the system can be even more productive.

5. Draw Mesh Instanced Indirect showed both better performance (about 20%) and more convenient use together with a compute shader than Draw Mesh Instanced.

If using a compute shader is not possible, the most efficient way to render on the CPU would be:

1) Calculate the position of the object in the shader (the Transform node in ASE is an example).
2) Disable Batching.
3) Deferred rendering.

4) Draw Mesh Instanced.

While the number of triangles does not increase with the number of particles, the complexity of the mesh will affect FPS. At 250 000 particles, 25 triangles in a mesh will give you 66 FPS, and 130 triangles will give you 33 FPS. Visual demonstration of the result of the article (Figure 7) [18].

The CPU to maintain 30 FPS is capable of handling no more than 2000 particles and 1 bot. The GPU, on the other hand, can handle millions of particles and a dozen bots for the same FPS. Consequently, the overall performance gain was $\sim 50\,000\%$, and using the GPU for parallel calculations opens up a huge computational opportunity from tessellating the snow underfoot to generating planets. The hypothesis of efficient use of compute shaders for adaptive particle systems has been fully confirmed.

Acknowledgements

The development of the system was commissioned and financed by Pingle Studio – Game Development Outsourcing Company.

[1] Living Particles. https://assetstore.unity.com/packages/vfx/particles/spells/living-particles-105817.

[2] Particle system. https://docs.unity3d.com/ScriptReference/ParticleSystem.html.

[3] Brodtkorb A. R., Hagen T. R., Shulz C., Hasle G. GPU computing in discrete optimization. Part I: Introduction to the GPU. EURO journal on transportation and logistics. **2** (1–2), 129–157 (2013).

[4] Brodtkorb A. R., Hagen T. R., Shulz C., Hasle G. GPU computing in discrete optimization. Part II: Survey focused on routing problems. EURO journal on transportation and logistics. **2** (1–2), 159–186 (2013).

[5] Dickinson C. Unity 2017 Game Optimization: Optimize all aspects of Unity performance. Birmingham, Packt Publishing Ltd. (2017).

[6] Anderdahl J., Darner A. Particle Systems Using 3D Vector Fields with OpenGL Compute Shaders. Computer Sciences Human Computer Interaction. Faculty of Computing Blekinge Institute of Technology, Karlskrona, Sweden (2014).

[7] Zeler W., Rohleder P. Particle effect system for the needs of a modern video game using the GPU. Machine graphics and vision. **25** (1/4), 35–44 (2016).

[8] Ježek B., Borecký J., Slabý A. Real time simulation and visualization of particle systems on GPU. AVR 2019: Augmented Reality, Virtual Reality, and Computer Graphics. 105–119 (2019).

[9] Stefan P. Particle system rendering: The effect on rendering speed when using geometry shaders. Bachelor thesis in Computer Science, May 2007. Department of Interaction and System Design. (2007).

[10] Va H., Choi M.-H., Hong M. Real-time cloth simulation using compute shader in Unity3D for AR/VR contents. Applied Sciences. **11** (17), 8255 (2021).

[11] Junker A., Palamas G. Real-time interactive snow simulation using compute shaders in digital environments. ACM international conference proceeding series. **70**, 1–4 (2020).

[12] Coding Adventure: Terraforming. https://www.youtube.com/watch?v=vTMEdHcKgM4&t=792s.

[13] Fu S., Sun S., Wang X., Liuef D. A brief overview of kernel methods with prior information. Procedia Computer Science. **199**, 269–275 (2022).

[14] Zamata S. M. E., Solis P. Y. M. Comparative of Techniques: Activation by Sequence, Morph Target Animation and CG/HLSL Programming in Surgery Incision Simulation for Virtual Reality. ICCIP '20: Proceedings of the 6th International Conference on Communication and Information Processing. 79–88 (2020).

[15] Halabi O., Khattak G. Generating haptic texture using solid noise. Displays. **69**, 102048 (2021).

[16] Polyakov V. Light performance comparison between forward, deferred and tile-based forward rendering. Blekinge, Blekinge Institute of Technology (2020).

[17] Ultimate Epic Battle Simulator 2.
https://store.steampowered.com/app/1468720/Ultimate_Epic_Battle_Simulator_2/.

[18] Optimized Ultimate Particle System. https://www.youtube.com/watch?v=xxGqBMI4zjI.

Використання обчислювального шейдеру для адаптивної системи частинок

Онуфрієнко Д. М.

Провідний технічний художник, Pingle Studio, Дніпро, Україна

У статті доведена гіпотеза високої ефективності використання обчислювального шейдера для системи частинок, здатної відстежувати інші об'єкти у просторі та адаптуватися до них у ігровому середовищі. Надано порівняльну характеристику ефективності роботи адаптивної системи частинок на обчислювальних основах CPU та GPU з додатковими методами оптимізації.

Ключові слова: *обчислювальний шейдер; GPU; система частинок; рендерінг; Unity; VFX; відео–гра.*