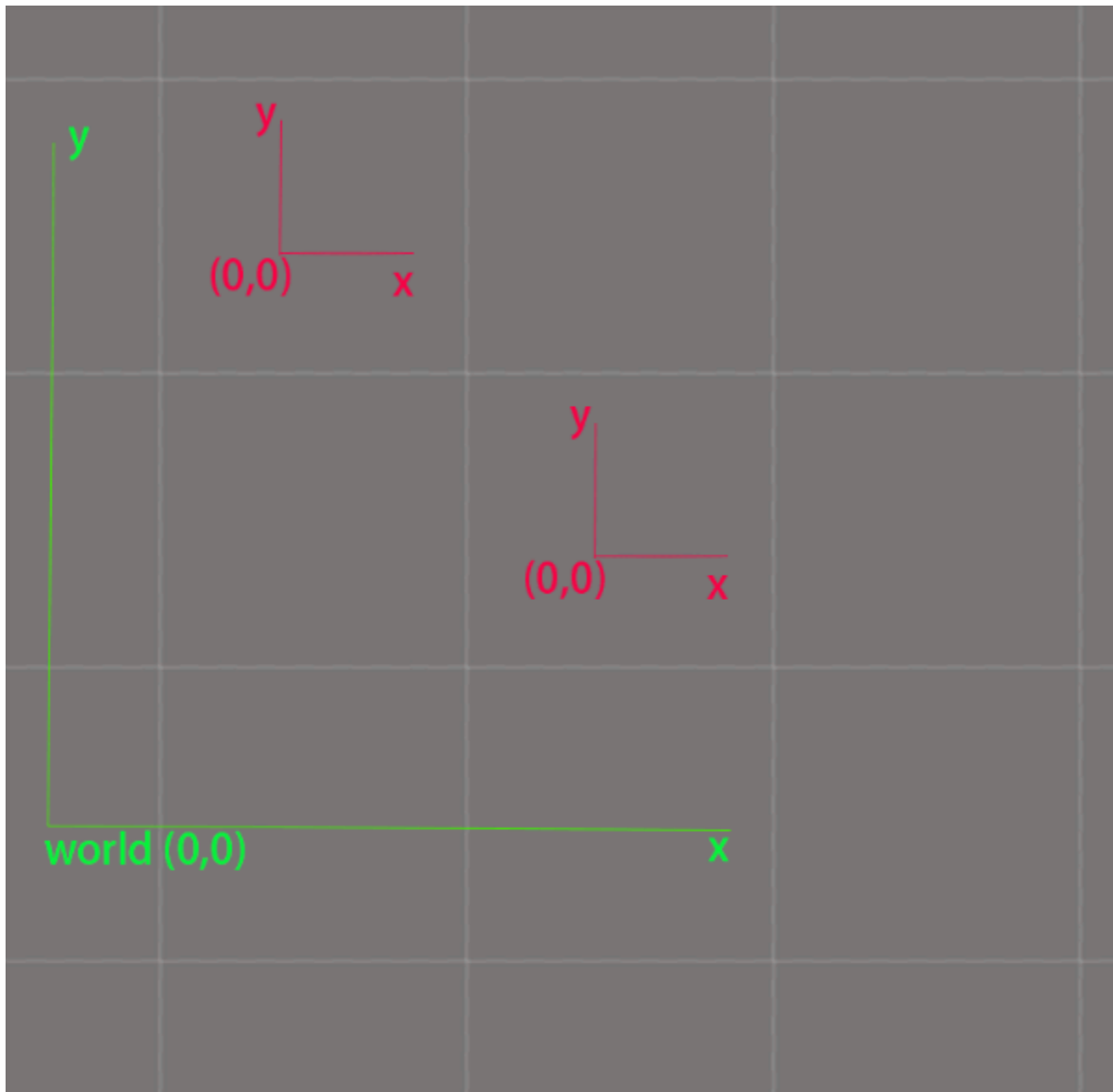


当世界足够大的时候，会导致浮点数的精度问题：

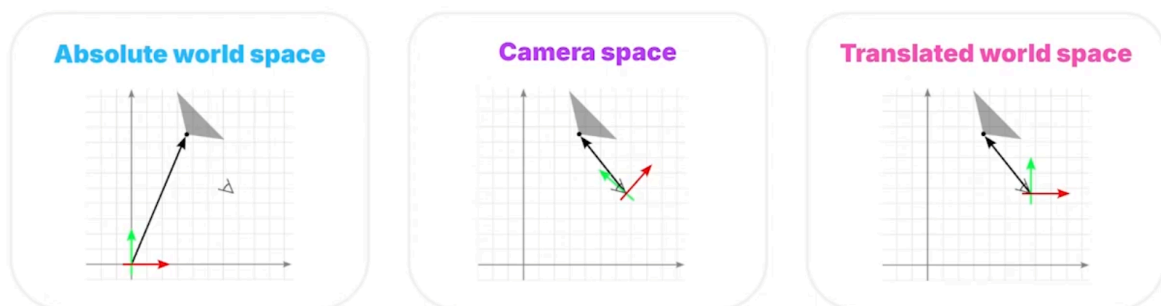


sovl 1

## 1. Translated world space

Store positions relative to the camera, or a point nearby.

- Perfect rendering precision: smallest vectors near camera
- Perfect performance: uses float32



采用此方法时，为了让物体能在相机范围内可见，每一帧都需要更新所有物体（会产生诸多问题：要考虑到渲染对象（objects to render），GPU compute structures(instance culling grids实例剔除网格)，space GI caches and stuff(世界空间全局光照缓存)) 当存在多相机时，还要考虑如何处理分屏和渲染目标（每个相机每帧 处理）还要考虑支持材质内容。

the solve :

## 2. Tiling

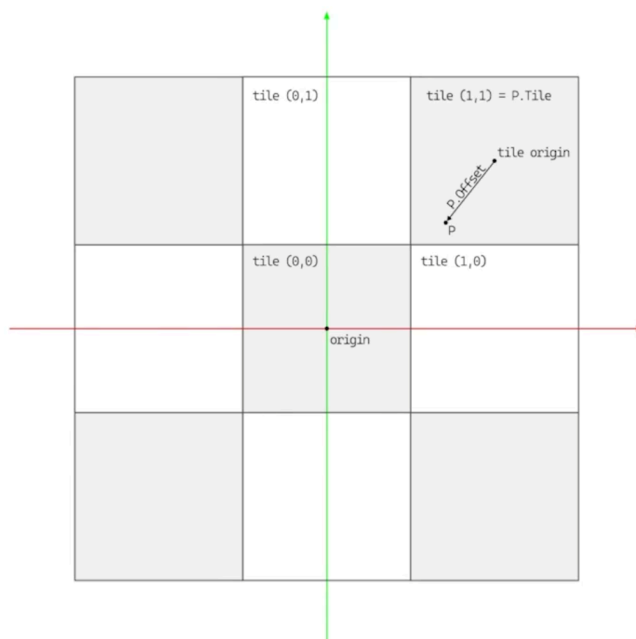
Take the world we had in UE4, and tile it.

Position P is encoded as

P: (Tile, Offset)

so that

$P = \text{Tile} * \text{TILE\_SIZE} + \text{Offset}$



## Math

Add:  $(T1, O1) + (T2, O2) = (T1 + T2, O1 + O2)$

Multiply (with float):  $(T, O) * F = (T * F, O * F)$

Multiply:  $(T1, O1) * (T2, O2) = (T1 * (T2 * \text{TILE\_SIZE} + O2), O1 * O2)$

Demote:  $T * \text{TileSize} + O$

Only translation matters for transforms

该方法对较小世界情况使用 tile it repat it in a sort of tiling fashion

当该点在世界中移动时并越过了世界的边缘，只需增加Tile值然后重置Offset，重复此步骤

只在shader和材质中进行相关计算

# Performance

- ~2x math instructions
- 2x memory usage
- But, can store only Offset for points in the same Tile

More expensive than camera-relative, but much cheaper than doubles

缺点：只适用于双精度条件下进行的操作

双精度在GPU计算会导致内存计算问题，会增加VRAM和内存器的内存成本。

we can actually optimize memory usage quite a bit by sharing the tile value across multiple different points if we know that they lie close together

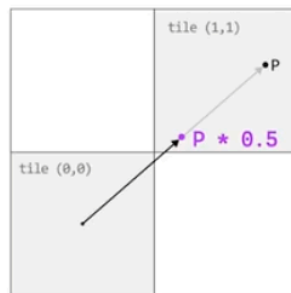
对于一个粒子系统，可以确定所有的粒子都相互靠近：可以上传一个tile value，然后隐式地将该tile value 共享，来给所有that upload for the specific individual particles

## Math problems

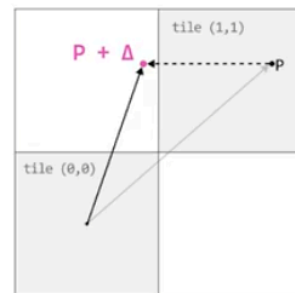
- Operations produce non-integer tile indices
  - Causes precision loss
  - May require 'regularization'



Tile = (1.0, 1.0)  
Offset = (0.6, 0.6)



Tile = (0.5, 0.5)  
Offset = (0.3, 0.3)



Tile = (1.0, 1.0)  
Offset = (-1.1, 0.6)

而对于向量p的计算过程中也存在精度问题，当对P进行\*0.5的操作时，计算后的值是Tile(0.5, 0.5)，而实际上在图中，P显然在tile(1, 1)中

同样，对P进行加法运算，也是如此

# Precision problems

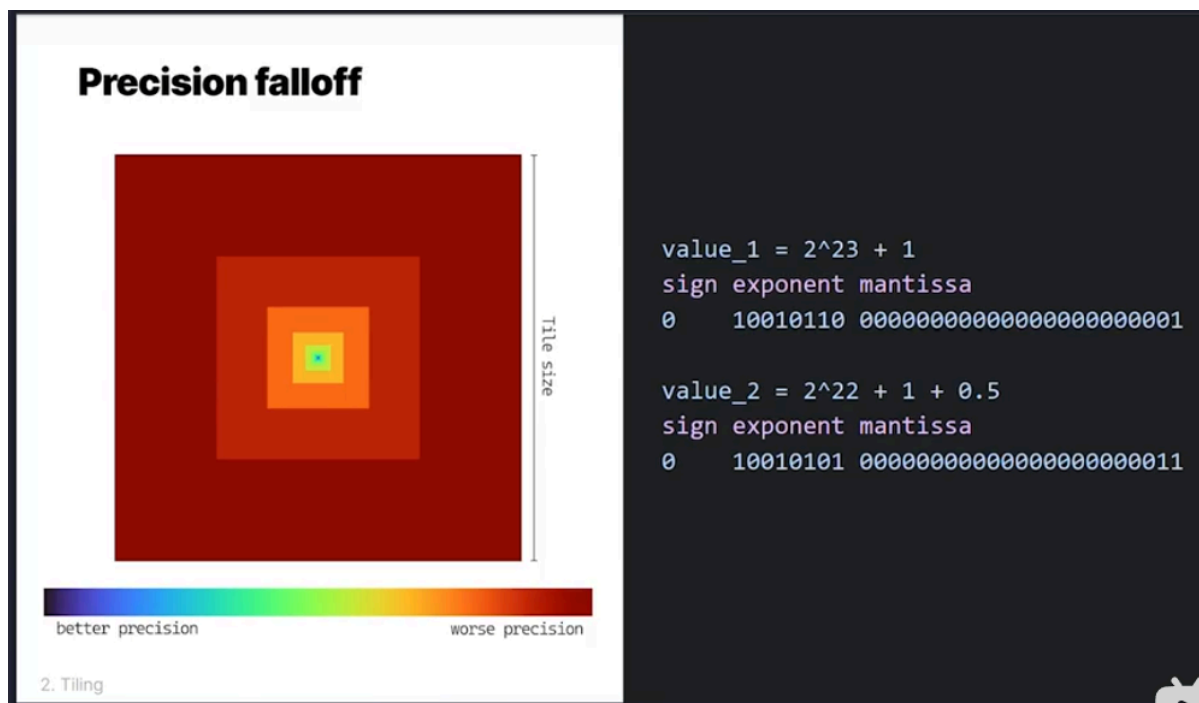
UE4 had  $2^{-4}$  precision at world edge.

So there is  $2^{-4}$  precision at the tile edge.

Turns out this is not enough.

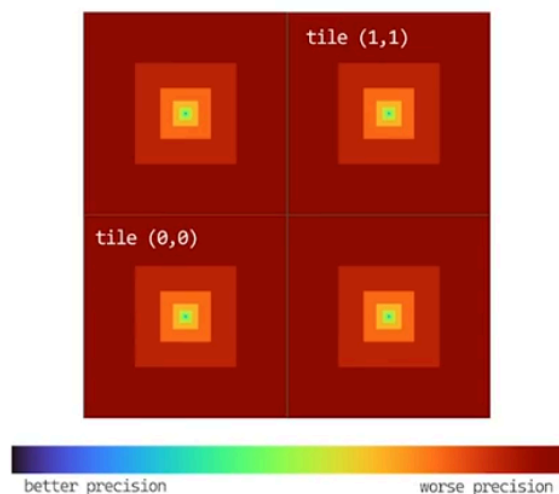
⇒ Shrink tiles?

another way : shrink the tile to avoid the precision ? - against the particle system



## Precision oscillation

Precision ramps up and down per tile



double float way to improve precision :

# Compared to float32 & float64

	Sign bits	Exponent bits	Precision (Mantissa bits)
float32	1	8	24 (23 explicit)
float64	1	11	53 (52 explicit)
DF	2*	16**	48 (46 explicit)***

## Math

1. Perform operation
2. Apply inverse operation
3. Calculate error

```
DFScalar TwoSum(float Lhs, float Rhs)
{
    // Do the operation that induces error
    float S = Lhs + Rhs // = Lhs + Rhs + Error

    // Revert operation to determine error
    float V = S - Lhs // = Rhs + Error1 caused by lhs>rhs
    float Q = S - V // = Lhs + Error2 by rhs>lhs

    float R = Lhs - Q // = Error2
    float T = Rhs - V // = Error1

    float E = R + T // = Error1+Error2

    // Return result with error, and a correction term
    return DFScalar(S, E)
}
```

ser: when use TwoSum need : 6 instruction , although better than doubles , but expensive ;

diable fast math :it may due to the etc precision be cut

## Summary

	Translated World Space	Tiling	DoubleFloat
Precision	✓	✗	✓
Performance	✓	✓	✗
Versatility	✗	✓	✓

conclusion :

putting it all mix these

use translated worldspace whenever possible ,otherwise use df :

## Example: drawing geometry

### Upload interframe data CPU → GPU

```
float3[] Vertices  
DFVector3 ObjectWorldPosition  
float3x3 ObjectRotationAndScale  
DFVector3 CameraWorldPosition
```



### GPU vertex shader

```
float3 ObjectToCameraTranslation =  
    DFSubtract(ObjectWorldPosition, CameraWorldPosition);  
  
float3 CameraVertex =  
    mul(Vertex, ObjectRotationAndScale) + ObjectToCameraTranslation;
```

upload the vertices ,the object transform and camera transform , the vertices just remain in single precision, it is just the mesh position and the camera position that we upload as double float vectors mesh rotation and scale remain as a single precision 3x3 matrix

on gpu ,can perform a single double float subtract operation(执行一次双精度浮点数减法计算操作), which subtracts the object from the camera position We get this relative vector from the camera to the object We know that this resulting vector will be small , because the object is close to the camera . then after the subtraction we can just bring that result down to single precision and then perform all the remaining steps in single precision.

只需要一个减法操作，在下一帧中，保留所有的mesh data cached 只需要重新上传新的相机位置。

## Example: drawing instanced geometry

Store instance position relative to root point,  
recombine to world position in shader

- 3 extra instructions
- Choose appropriate RootPosition

### Upload interframe data CPU → GPU

```
float[] Vertices  
float3 RootPosition  
float3[] InstancePosition  
float3x3[] InstanceRotationAndScale  
DFVector3 CameraWorldPosition
```



### GPU vertex shader

```
DFVector3 InstanceWorldPosition = DFFastTwoSum(RootPosition, InstancePosition[Index]);  
float3 LocalToCameraTranslation =  
    DFSubtract(InstanceWorldPosition, CameraWorldPosition);  
  
float3 CameraVertex =  
    mul(Vertex, InstanceRotationAndScale[Index]) + LocalToCameraTranslation;
```

## DF Subtract

20 instructions, but we can do better

```
DFScalar Subtract(DFScalar Lhs, DFScalar Rhs)
{ return Add(Lhs, -Rhs) }

DFScalar Add(DFScalar Lhs, DFScalar Rhs)
{

    DFScalar S = TwoSum(Lhs.High, Rhs.High) // 6 additions
    DFScalar T = TwoSum(Lhs.Low, Rhs.Low) // 6 additions

    // Merge and rebalance
    S.Low += T.High
    S = FastTwoSum(S.High, S.Low) // 3 additions
    S.Low += T.Low
    S = FastTwoSum(S.High, S.Low) // 3 additions

    return S

}
```