**基于shape-e的三维模型生成及雪花飘落运动算法手动实现**

**1.shape-e介绍**

Shape-E用于学习和生成3D形状。

这个算法用到了OpenAI的模型Clip的概念，将图像和文本描述结合起来以生成3D形状。它基于一个简单的想法： 将形状编码为一个3D张量，然后用一个自编码器来学习这个编码。

在训练过程中，Shape-E使用了两个数据集： ShapeNet和命令行生成的3D形状。在ShapeNet数据集中，每个物体都由一个3D模型和若干视图组成，每个视图都配有一个描述。在命令行生成的3D形状数据集中，每个物体都由一个3D模型，若干视图和一个命令行描述组成。

Shape-E的工作流程是这样的：

* 先在数据集上运行自编码器以学习形状的3D编码。
* 使用Clip模型，将每个形状视图的文本描述编码成一个向量。
* 将这个向量和形状的3D编码结合起来，使用一个生成器生成新的3D形状。

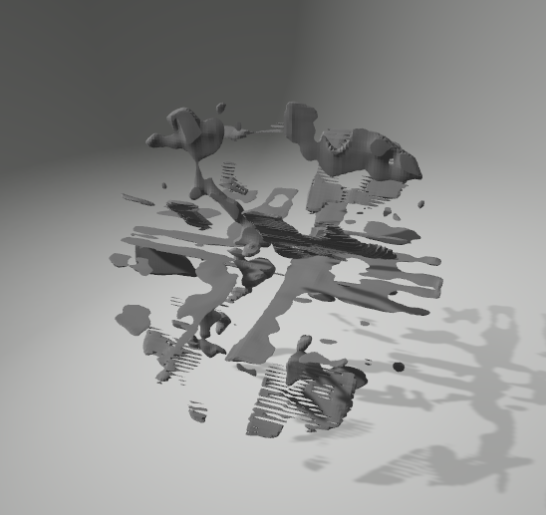
Shape-E的一个重要特点是，它可以处理非常复杂的形状，包括那些具有精细结构的形状。

**2.雪粒子生成**

使用图生obj功能，识别雪花图片中的雪花主体，生成雪花obj模型。

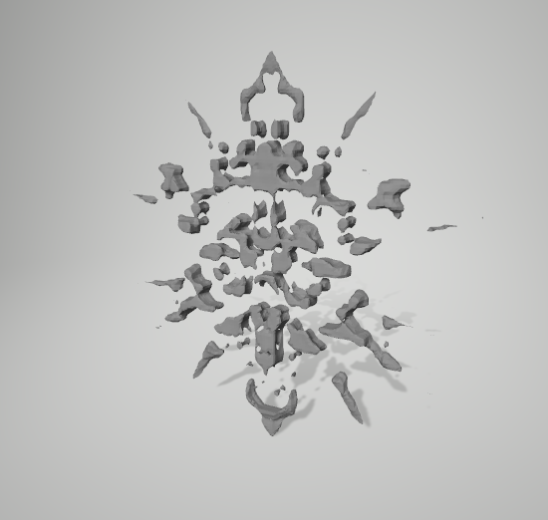
雪花主体的png格式文件。

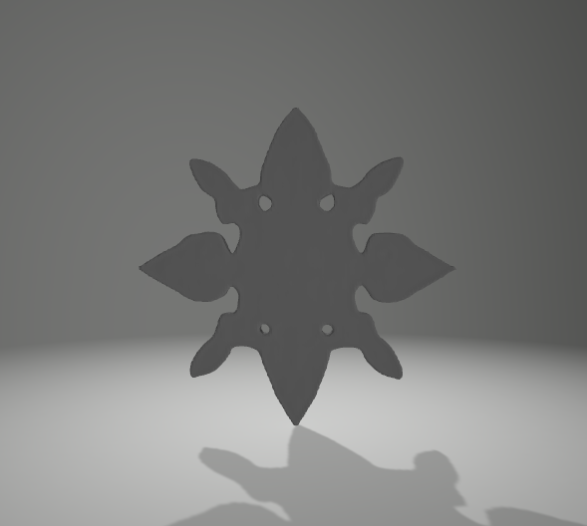
**Image to 3D obj,将产生具有深度空间的球形雪花。**

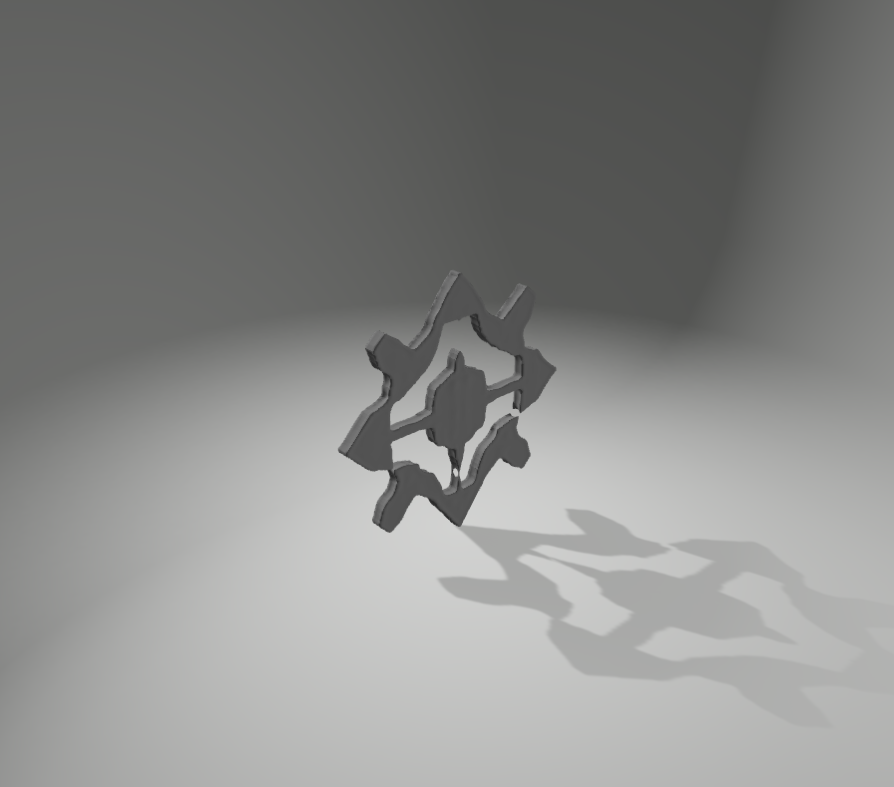


**Text to 3D obj, 提示词输入"a hexagon snowflake"，将产生六边形平面状雪花。**









**2.verlet算法原理和优势**

Verlet积分算法是一种用于在计算机模拟中解决牛顿运动方程的方法。该算法在计算机图形学，特别是在物理引擎和粒子系统模拟中得到了广泛的应用。

Verlet算法的基本形式是这样的：

newPosition = 2\*currentPosition - oldPosition + acceleration \* deltaTime^2

其中，currentPosition 是物体当前的位置，oldPosition 是物体在上一时间步的位置，acceleration 是物体当前受到的加速度（可以由力/质量得到），deltaTime 是时间步长。

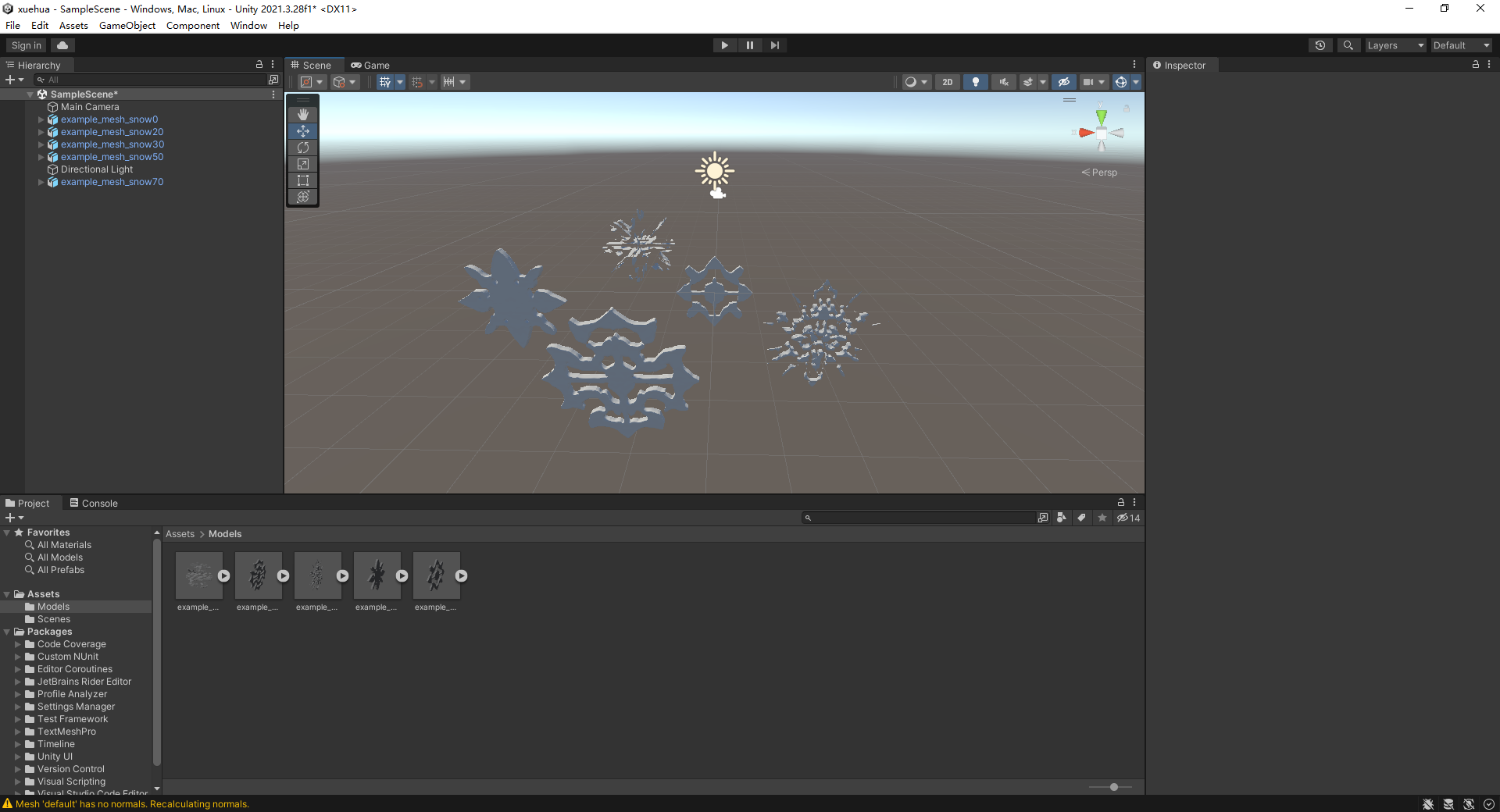
这个算法的重要优点之一是它在屏幕上产生稳定可靠的运动，而且不受时间步长的影响，在计算相同的物理场景时，无论时间步长如何变化，结果总是一致的。

另一个重要的优点是它对于模拟系统中的能量守恒表现出非常好的性能。在很多物理模拟中，我们希望动能和势能的总和（也就是总能量）在系统演化过程中是保持不变的。然而，在一些其他的数值积分方法中，由于数值误差的累积，总能量会逐渐偏离真实值，而Verlet积分算法能够非常好地维持总能量的守恒。

综合上述优点，Verlet积分算法因此成为了很多物理模拟，特别是在游戏中的首选算法。

**3.unity 实践**

shape-e生的obj放入到unity工程目录中



为了实现雪花飘落运动的模拟，要考虑的因素较多，包括重力、风力、空气阻力、湍流、涡旋和雪花旋转。由于Unity不直接支持Verlet积分，我们需要自己实现。

**积分代码：**

|  |
| --- |
| JSON public class VerletIntegrator {  private Vector3 oldPosition;  private Vector3 acceleration;  private float drag = 0.995f;   public VerletIntegrator(Vector3 initialPosition, Vector3 initialVelocity, Vector3 initialAcceleration)  {  oldPosition = initialPosition - initialVelocity \* Time.fixedDeltaTime;  acceleration = initialAcceleration;  }   public Vector3 Integrate(Vector3 position)  {  Vector3 tempPos = position;  position = position + (position - oldPosition) \* drag + acceleration \* Time.fixedDeltaTime \* Time.fixedDeltaTime;  oldPosition = tempPos;  return position;  }   public void AddForce(Vector3 force)  {  acceleration += force;  } } |

* VerletIntegrator(Vector3 initialPosition, Vector3 initialVelocity, Vector3 initialAcceleration): 构造器，初始化oldPosition, acceleration。
* Integrate(Vector3 position): 进行一次Verlet积分计算，更新position和oldPosition，并返回计算后的新position。
* AddForce(Vector3 force): 给当前物体添加一个力，它会被加到acceleration上。
* oldPosition是物体在上一帧的位置；
* acceleration是物体的加速度，起初被设为一个初始值，然后每当调用AddForce时，该力就会被加到acceleration上。
* drag代表空气阻力或摩擦力，影响物体的速度。

在Integrate函数中，它先把当前位置存储到tempPos中，然后计算新的位置，这个新的位置等于当前位置加上当前位置和旧位置的差（即速度）乘以阻力（模拟摩擦）再加上加速度乘以时间的平方（基于基本的物理公式：s = ut + 1/2at^2）。然后，它把旧的位置设置为tempPos（即更新旧的位置为当前位置），并返回新的位置。

**柏林噪声：**

雪花在x和z方向上使用柏林随机噪声产生一个随机横向力以模拟雪花随风飘动，并将该力和重力一起作用于verlet积分上。

|  |
| --- |
| JSON Vector3 perlinDisplacement = new Vector3((Mathf.PerlinNoise(Time.time\*0.1f, 0) \* 1.0f - 0.5f), 0, (Mathf.PerlinNoise(0, Time.time\*0.1f) \* 1.0f - 0.5f)); integrator.AddForce(gravity + perlinDisplacement); transform.position = integrator.Integrate(transform.position); |

**完整demo例子：**

|  |
| --- |
| JSON using System.Collections; using System.Collections.Generic; using UnityEngine; using UnityEngine.UIElements;  public class VerletIntegrator {  private Vector3 oldPosition;  private Vector3 acceleration;  private float drag = 0.995f;   public VerletIntegrator(Vector3 initialPosition, Vector3 initialVelocity, Vector3 initialAcceleration)  {  oldPosition = initialPosition - initialVelocity \* Time.fixedDeltaTime;  acceleration = initialAcceleration;  }   public Vector3 Integrate(Vector3 position)  {  Vector3 tempPos = position;  position = position + (position - oldPosition) \* drag + acceleration \* Time.fixedDeltaTime \* Time.fixedDeltaTime;  oldPosition = tempPos;  return position;  }   public void AddForce(Vector3 force)  {  acceleration += force;  } } public class Snowflake : MonoBehaviour {  public Vector3 gravity = new Vector3(0, -0.0981f, 0); // Gravity  private VerletIntegrator integrator;  private Quaternion rot;  private float changeRotationInterval = 1.0f; // The interval between each rotation change, in seconds.  private float timeSinceLastRotationChange;   void Start()  {  timeSinceLastRotationChange = 0;  integrator = new VerletIntegrator(transform.position, Vector3.zero, Vector3.zero);  integrator.AddForce(gravity);  rot = Quaternion.Euler(Random.Range(-360, 360), Random.Range(-360, 360), Random.Range(-360, 360));  }   void FixedUpdate()  {  // Adding perlin noise for random displacement in x and z direction  Vector3 perlinDisplacement = new Vector3((Mathf.PerlinNoise(Time.time, 0) \* 1.0f - 0.5f), 0, (Mathf.PerlinNoise(0, Time.time) \* 1.0f - 0.5f));   // Updating the position integrating the forces  // Add a force that pulls the particle back towards the origin   // Add forces to the integrator  integrator.AddForce(gravity + 2.0f\*perlinDisplacement);  transform.position = integrator.Integrate(transform.position);   timeSinceLastRotationChange += Time.fixedDeltaTime;   if (timeSinceLastRotationChange >= changeRotationInterval)  {  rot = Quaternion.Euler(Random.Range(-360, 360), Random.Range(-360, 360), Random.Range(-360, 360)); // Update rotation  timeSinceLastRotationChange = 0;  }   transform.rotation = Quaternion.Lerp(transform.rotation, rot, Time.fixedDeltaTime);  } } |

雪花运动：

