Video Link:
Textured Cone-shaped particle system
https://drive.google.com/file/d/18osPfOqwQ8dYcGVInaIWo8P7GfzaRz_J/view?usp=drive_link
Single Colored Sphere-shaped particle system
https://drive.google.com/file/d/1pjQpV5dHRdxdXu8penSyjdtwcRv3Aqj5/view?usp=drive_link

I implemented a particle system with two different shapes and two different materials
Under PrimeEngine/Scene, I created a particleSystem struct that inherits Mesh

```
struct ParticleSystem: Mesh
{
    PE_DECLARE_CLASS(ParticleSystem);
    // Constructor -----------------------------------------------------------
    ParticleSystem(PE::GameContext &context, PE::MemoryArena arena, Handle hMyself);

    virtual ~ParticleSystem() {}

    // Component -------------------------------------------------------------

    virtual void addDefaultComponents();

    void createParticleSystem(Particle pTemplate);

    void loadParticle_needsRC(int &threadOwnershipMask);

    PE_DECLARE_IMPLEMENT_EVENT_HANDLER_WRAPPER(do_GATHER_DRAWCALLS);
    virtual void do_GATHER_DRAWCALLS(Events::Event *pEvt);

    PE_DECLARE_IMPLEMENT_EVENT_HANDLER_WRAPPER(do_UPDATE);
    virtual void do_UPDATE(Events::Event *pEvt);

    Handle m_hParticleSystemCPU;

    Handle m_meshCPU;

    Matrix4x4 m_offset;

    PrimitiveTypes::Bool m_loaded;

    PrimitiveTypes::Bool m_hasTexture;
    PrimitiveTypes::Bool m_hasColor;

    PE::MemoryArena m_arena; PE::GameContext *m_pContext;
```

When Initializing the component, I added the particle system under RootSceneNode to display the effect. During Initialization, I registered Update and Gather_drawcall event for this

component.

```
namespace PE
{
    namespace Components
    {
        PE_IMPLEMENT_CLASS1(ParticleSystem, Mesh);
        ParticleSystem::ParticleSystem(PE::GameContext &context, PE::MemoryArena arena, Handle hMyself)
            :Mesh([&]context, arena, hMyself)
        {
            m_offset = Matrix4x4();
            m_offset.setPos(p: Vector3(x: 0, y: 0, z: 0));
            m_arena = arena;
            m_pContext = &context;
        }

        void ParticleSystem::addDefaultComponents()
        {
            Mesh::addDefaultComponents();
            PE_REGISTER_EVENT_HANDLER(Events::Event_GATHER_DRAWCALLS, ParticleSystem::do_GATHER_DRAWCALLS);
            PE_REGISTER_EVENT_HANDLER(Events::Event_UPDATE, ParticleSystem::do_UPDATE);
        }
```

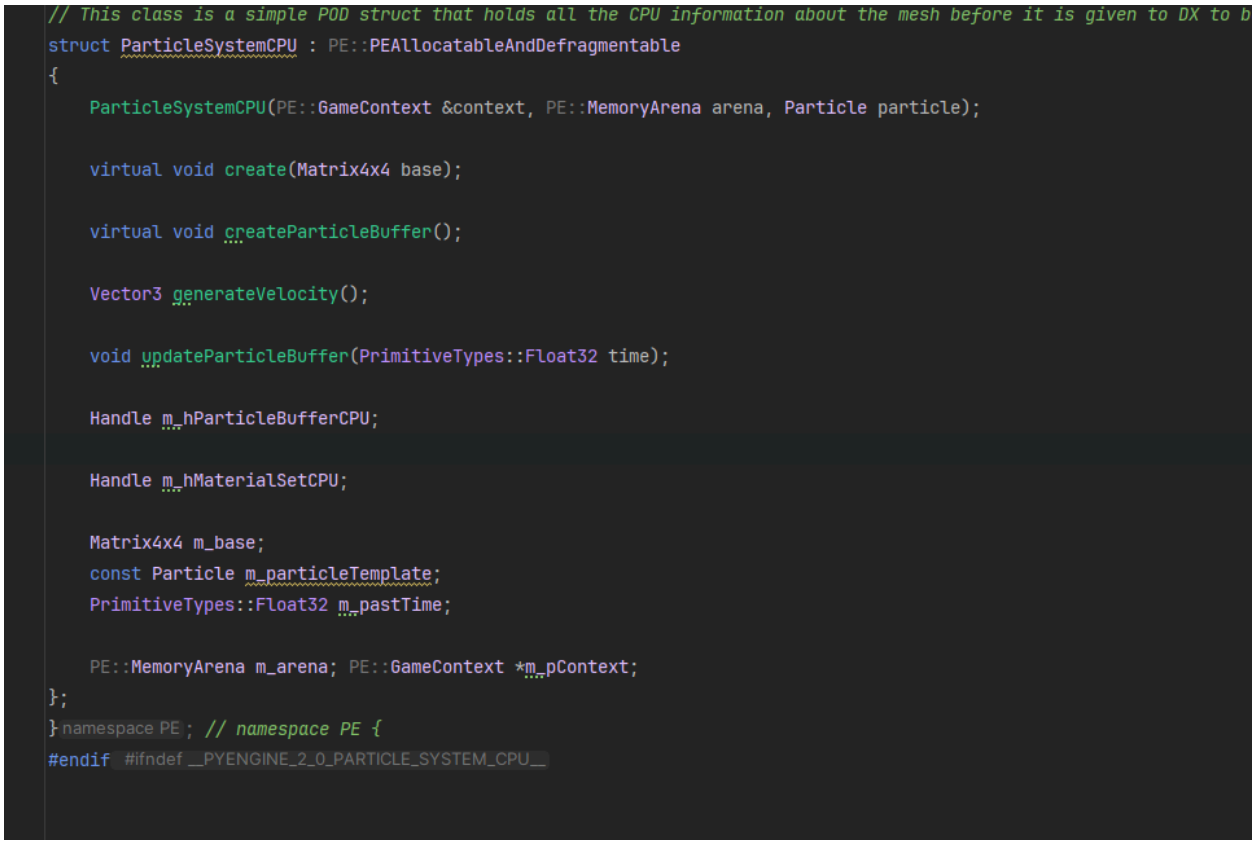
I set a template for each particle during initialization

```
        void ParticleSystem::createParticleSystem(Particle pTemplate)
        {
            if (!m_hParticleSystemCPU.isValid())
            {
                m_hParticleSystemCPU = Handle(dbgName: "PARTICLESYSTEMCPU", neededSize: sizeof(ParticleSystemCPU));
                new (m_hParticleSystemCPU) ParticleSystemCPU([&]*m_pContext, m_arena, pTemplate);
            }
            ParticleSystemCPU &pSysCPU = *m_hParticleSystemCPU.getObject<ParticleSystemCPU>();

            Handle h_parent = getFirstParentByType<SceneNode>();
            if (h_parent.isValid())
            {
                SceneNode* pSN = h_parent.getObject<SceneNode>();
                Matrix4x4 particleBase = pSN->m_worldTransform * m_offset;
                m_hasTexture = strlen(pTemplate.m_texture) > 0;
                m_hasColor = (pTemplate.color.m_x != 0 && pTemplate.color.m_y != 0 && pTemplate.color.m_z != 0);
                // m_hasColor = true;

                pSysCPU.create(particleBase);
            } else
            {
                OutputDebugStringA("no parent Scene node");
            }
        }
```

```
struct Particle
{
    PrimitiveTypes::Int16 m_rate;
    PrimitiveTypes::Float32 m_speed;
    const PrimitiveTypes::Float32 m_duration;
    PrimitiveTypes::Bool m_looping;
    Vector2 m_size;
    Shape m_shape;

    const char* m_texture;
    Vector3 color;
};
```

```
void ParticleSystemCPU::createParticleBuffer()
{
    m_hParticleBufferCPU = Handle( dbgName: "PARTICLE_BUFFER_CPU", neededSize: sizeof(ParticleBufferCPU<ParticleCPU>));
    ParticleBufferCPU<ParticleCPU> *ppbcpu = new(m_hParticleBufferCPU) ParticleBufferCPU<ParticleCPU>( [&] *m_pContext, m_arena);
    const PrimitiveTypes::Int32 maxParticleSize = m_particleTemplate.m_duration * m_particleTemplate.m_rate;
    ppbcpu->m_values.reset(maxParticleSize);

    // m_hTexCoordBufferCPU = Handle("TEXCOORD_BUFFER_CPU", sizeof(TexCoordBufferCPU));
    // TexCoordBufferCPU *ptcbcpu = new(m_hTexCoordBufferCPU) TexCoordBufferCPU();
    // ptcbcpu->m_values.reset(nsprites * 2);


    ParticleCPU newParticle = { .m_base: Matrix4x4(),m_particleTemplate.m_size, .velocity: m_base.getN(), .m_age: m_particleTemplate.m_duration};
    ppbcpu->m_values.add(newParticle);

    m_hMaterialSetCPU = Handle( dbgName: "MATERIAL_SET_CPU", neededSize: sizeof(MaterialSetCPU));
    MaterialSetCPU *pmscpu = new(m_hMaterialSetCPU) MaterialSetCPU( [&] *m_pContext, m_arena);
    pmscpu->createSetWithOneTexturedMaterial(m_particleTemplate.m_texture, package: "Default");
}

void ParticleSystemCPU::create(Matrix4x4 base)
{
    m_base = Matrix4x4(base);
    createParticleBuffer();
}
```

Inside ParticleSystemCPU, I define the data for the scene to render, basically setting up the template and current particle positions.



```
// This class is a simple POD struct that holds all the CPU information about the mesh before it is given to DX to b
struct ParticleSystemCPU : PE::PEAllocatableAndDefragmentable
{
    ParticleSystemCPU(PE::GameContext &context, PE::MemoryArena arena, Particle particle);

    virtual void create(Matrix4x4 base);

    virtual void createParticleBuffer();

    Vector3 generateVelocity();

    void updateParticleBuffer(PrimitiveTypes::Float32 time);

    Handle m_hParticleBufferCPU;

    Handle m_hMaterialSetCPU;

    Matrix4x4 m_base;
    const Particle m_particleTemplate;
    PrimitiveTypes::Float32 m_pastTime;

    PE::MemoryArena m_arena; PE::GameContext *m_pContext;
};
} namespace PE ; // namespace PE {
#endif  #ifndef __PYENGINE_2_0_PARTICLE_SYSTEM_CPU__
```

During update event, the frametime is passed in and updates the particleBuffer.

```
void ParticleSystemCPU::updateParticleBuffer(PrimitiveTypes::Float32 time)
{
    ParticleBufferCPU<ParticleCPU> *ppbcpu;
    if (!m_hParticleBufferCPU.isValid())
    {
        m_hParticleBufferCPU = Handle( dbgName: "PARTICLE_BUFFER_CPU",  neededSize: sizeof(ParticleBufferCPU<ParticleCPU>));
        ppbcpu = new(m_hParticleBufferCPU) ParticleBufferCPU<ParticleCPU>( [&] *m_pContext, m_arena);
        const PrimitiveTypes::Int32 maxParticleSize = m_particleTemplate.m_duration * m_particleTemplate.m_rate;
        ppbcpu->m_values.reset(maxParticleSize);
    } else
    {
        ppbcpu = m_hParticleBufferCPU.getObject<ParticleBufferCPU<ParticleCPU>>();
    }
    m_pastTime += time;
    int totalParticles = m_pastTime * m_particleTemplate.m_rate;

    for (int j = 0; j < ppbcpu->m_values.m_size; j++)
    {
        ppbcpu->m_values[j].m_age -= time;
        if (ppbcpu->m_values[j].m_age <= 0)
        {
            ParticleCPU newParticle = { .m_base: Matrix4x4(),
                m_particleTemplate.m_size,
                generateVelocity(), .m_age: m_particleTemplate.m_duration};
            ppbcpu->m_values[j] = newParticle;
        } else
        {
            Vector3 curPos = ppbcpu->m_values[j].m_base.getPos();
            curPos += time * ppbcpu->m_values[j].velocity;
            ppbcpu->m_values[j].m_base.setPos(curPos);
        }
    }
```

When updating the particle buffer, I also set the rotation of each particle to face the camera but manually setting the u, n and v direction.

```
    }
    if (totalParticles < ppbcpu->m_values.m_capacity && totalParticles > ppbcpu->m_values.m_size)
    {
        PrimitiveTypes::Int16 partCount = totalParticles - ppbcpu->m_values.m_size;
        for (int i = 0; i < partCount; i++)
        {
            ParticleCPU newParticle = { .m_base: Matrix4x4(),
                m_particleTemplate.m_size,
                generateVelocity(), .m_age: m_particleTemplate.m_duration};
            ppbcpu->m_values.add(newParticle);
        }
    }
    for (int j = 0; j < ppbcpu->m_values.m_size; j++)
    {
        Components::CameraSceneNode *pcam = Components::CameraManager::Instance()->getActiveCamera()->getCamSceneNode();
        Vector3 cameraFront = pcam->m_worldTransform.getN();
        Vector3 cameraRight = pcam->m_worldTransform.getU();
        Vector3 cameraUp = pcam->m_worldTransform.getV();

        ppbcpu->m_values[j].m_base.setN(cameraFront);
        ppbcpu->m_values[j].m_base.setU(cameraRight);
        ppbcpu->m_values[j].m_base.setV(cameraUp);
    }

}
```

For each shape of the particle system, I calculated the direction of velocity based on different shapes.

```
Vector3 ParticleSystemCPU::generateVelocity()
{
    PrimitiveTypes::Float32 angleRange = 0.f;
    Matrix4x4 tempMat;

    switch(m_particleTemplate.m_shape)
    {
        case Cone:
            {
                PrimitiveTypes::Float32 eularAnglex = (float)(rand() % 45) - (45.f / 2.f);
                PrimitiveTypes::Float32 radianx = (eularAnglex / 180.f) * PrimitiveTypes::Constants::c_Pi_F32;
                PrimitiveTypes::Float32 eularAngley = (float)(rand() % 45) - (45.f / 2.f);
                PrimitiveTypes::Float32 radiany = (eularAngley / 180.f) * PrimitiveTypes::Constants::c_Pi_F32;
                tempMat.turnRight(radianx);
                tempMat.turnUp(radiany);
                break;
            }

        case Sphere:
            {
                PrimitiveTypes::Float32 eularAnglex = (float)(rand() % 360) - (360.f / 2.f);
                PrimitiveTypes::Float32 radianx = (eularAnglex / 180.f) * PrimitiveTypes::Constants::c_Pi_F32;
                PrimitiveTypes::Float32 eularAngley = (float)(rand() % 360) - (360.f / 2.f);
                PrimitiveTypes::Float32 radiany = (eularAngley / 180.f) * PrimitiveTypes::Constants::c_Pi_F32;
                tempMat.turnRight(radianx);
                tempMat.turnUp(radiany);
                break;
            }
        default:
            {
                OutputDebugStringA("ParticleSystemCPU::createParticleBuffer(): current shape not supported");
                break;
```

Last, feed the position data into Particle system and generate vertex buffer, index buffer, texture coordinates and normal buffers if texture file exists, color buffer if color exists.

```
void ParticleSystem::loadParticle_needsRC(int &threadOwnershipMask)
{
    MeshCPU* mcpu;
    if (!m_meshCPU.isValid())
    {
        m_meshCPU = Handle( dbgName: "MeshCPU SpriteMesh",  neededSize: sizeof(MeshCPU));
        mcpu = new (m_meshCPU) MeshCPU( [&] *m_pContext, m_arena);
    }
    else
    {
        mcpu = m_meshCPU.getObject<MeshCPU>();
    }
    if (!m_loaded)
    {
        mcpu->createEmptyMesh();
        // mcpu.createBillboardMeshWithColorTexture("cobble2_color.dds", "Default", 32, 32, SamplerState_NoMips_NoMinTex

    }
    mcpu->m_manualBufferManagement = true;
    ParticleSystemCPU* pSysCPU = m_hParticleSystemCPU.getObject<ParticleSystemCPU>();

    ParticleBufferCPU<ParticleCPU> *pPb = pSysCPU->m_hParticleBufferCPU.getObject<ParticleBufferCPU<ParticleCPU>>();
    PrimitiveTypes::Int16 particleCount = pPb->m_values.m_size;

    PositionBufferCPU *pVB = mcpu->m_hPositionBufferCPU.getObject<PositionBufferCPU>();
    IndexBufferCPU *pIB = mcpu->m_hIndexBufferCPU.getObject<IndexBufferCPU>();
```

```
PositionBufferCPU *pVB = mcpu->m_hPositionBufferCPU.getObject<PositionBufferCPU>();
IndexBufferCPU *pIB = mcpu->m_hIndexBufferCPU.getObject<IndexBufferCPU>();

ColorBufferCPU *pCB;
TexCoordBufferCPU *pTCB;
NormalBufferCPU *pNB;
MaterialSetCPU *msCPU;

pVB->m_values.reset( capacity: particleCount * 4 * 3); // 4 verts * (x,y,z)
pIB->m_values.reset(capacity: particleCount * 6); // 2 tris

pIB->m_indexRanges[0].m_start = 0;
pIB->m_indexRanges[0].m_end = particleCount * 6 - 1;
pIB->m_indexRanges[0].m_minVertIndex = 0;
pIB->m_indexRanges[0].m_maxVertIndex = particleCount * 4 - 1;

pIB->m_minVertexIndex = pIB->m_indexRanges[0].m_minVertIndex;
pIB->m_maxVertexIndex = pIB->m_indexRanges[0].m_maxVertIndex;
if (m_hasTexture)
{
    pTCB = mcpu->m_hTexCoordBufferCPU.getObject<TexCoordBufferCPU>();
    pTCB->m_values.reset(capacity: particleCount * 4 * 2);

    pNB = mcpu->m_hNormalBufferCPU.getObject<NormalBufferCPU>();
    pNB->m_values.reset(capacity: particleCount * 4 * 3);
}
if (m_hasColor)
```

```
if (m_hasColor)
{
    pCB = mcpu->m_hColorBufferCPU.getObject<ColorBufferCPU>();
    pCB->m_values.reset(capacity: particleCount * 4 * 3);
}
for (int i = 0; i < particleCount; i++)
{
    Matrix4x4 topLeftTransform = pPb->m_values[i].m_base;
    Matrix4x4 topRightTransform = pPb->m_values[i].m_base;
    Matrix4x4 bottomLeftTransform = pPb->m_values[i].m_base;
    Matrix4x4 bottomRightTransform = pPb->m_values[i].m_base;
    Vector2 curSize = pPb->m_values[i].m_size;
    topLeftTransform.moveLeft(distance: (curSize.m_x / 2.f));
    topLeftTransform.moveUp(distance: (curSize.m_y / 2.f));

    topRightTransform.moveRight(distance: (curSize.m_x / 2.f));
    topRightTransform.moveUp(distance: (curSize.m_y / 2.f));

    bottomLeftTransform.moveLeft(distance: (curSize.m_x / 2.f));
    bottomLeftTransform.moveDown(distance: (curSize.m_y / 2.f));

    bottomRightTransform.moveRight(distance: (curSize.m_x / 2.f));
    bottomRightTransform.moveDown(distance: (curSize.m_y / 2.f));

    pVB->m_values.add(topLeftTransform.getPos().m_x, topLeftTransform.getPos().m_y, topLeftTransform.getPos().m_z); // top lef
    pVB->m_values.add(topRightTransform.getPos().m_x, topRightTransform.getPos().m_y, topRightTransform.getPos().m_z); // top right
    pVB->m_values.add(bottomRightTransform.getPos().m_x, bottomRightTransform.getPos().m_y, bottomRightTransform.getPos().m_z);
    pVB->m_values.add(bottomLeftTransform.getPos().m_x, bottomLeftTransform.getPos().m_y, bottomLeftTransform.getPos().m_z);

    pIB->m_values.add(i * 4 + 0, i * 4 + 1, i * 4 + 2);
```

```
        pVB->m_values.add(topRightTransform.getPos().m_x, topRightTransform.getPos().m_y, topRightTransform.getPos().m_z); // top righ
        pVB->m_values.add(bottomRightTransform.getPos().m_x, bottomRightTransform.getPos().m_y, bottomRightTransform.getPos().m_z);
        pVB->m_values.add(bottomLeftTransform.getPos().m_x, bottomLeftTransform.getPos().m_y, bottomLeftTransform.getPos().m_z);

        pIB->m_values.add(i * 4 + 0, i * 4 + 1, i * 4 + 2);
        pIB->m_values.add(i * 4 + 2, i * 4 + 3, i * 4 + 0);

        if (m_hasColor)
        {
            pCB->m_values.add(0, 1.f, 0);
            pCB->m_values.add(0, 1.f, 0);
            pCB->m_values.add(0, 1.f, 0);
            pCB->m_values.add(0, 1.f, 0);
        }

        if (m_hasTexture)
        {
            pTCB->m_values.add(0, 0); // top left
            pTCB->m_values.add(1, 0); // top right
            pTCB->m_values.add(1, 1);
            pTCB->m_values.add(0, 1);

            pNB->m_values.add(0, 0, 0);
            pNB->m_values.add(0, 0, 0);
            pNB->m_values.add(0, 0, 0);
            pNB->m_values.add(0, 0, 0);
        }
    }
```

```
    if (!m_loaded)
    {
        // first time creating gpu mesh
        if (m_hasTexture)
        {
            msCPU = mcpu->m_hMaterialSetCPU.getObject<MaterialSetCPU>();
            memcpy(msCPU, _Src: pSysCPU->m_hMaterialSetCPU.getObject<MaterialSetCPU>(), _Size: sizeof(msCPU));
        }
        loadFromMeshCPU_needsRC([&] *mcpu, [&] threadOwnershipMask);
        const char* techName = "StdMesh_Diffuse_Tech";
        if (techName && !m_hasColor)
        {
            Handle hEffect = EffectManager::Instance()->getEffectHandle(techName);

            for (unsigned int imat = 0; imat < m_effects.m_size; imat++)
            {
                if (m_effects[imat].m_size)
                    m_effects[imat][0] = hEffect;
            }
        }
        m_loaded = true;
    } else
    {
        updateGeoFromMeshCPU_needsRC([&] *mcpu, [&] threadOwnershipMask);
    }
}
```

Last, add them to the rootSceneNode, and load the render buffers whenever render context is available

```
PE::Handle pSHandle( dbgName: "PARTICLE_SYSTEM",  neededSize: sizeof(ParticleSystem));
ParticleSystem* pSys = new(pSHandle) ParticleSystem( [&] *m_pContext, m_arena,  pSHandle);
// //
Particle pTemplate = {
    .m_rate: 10,  .m_speed: 5.f,  .m_duration: 5.f,  .m_looping: true,  .m_size: Vector2( x: .1f,  y: .1f),
    Shape::Cone,  .m_texture: "necronaut.dds",  .color: Vector3( x: 0,  y: 0,  z: 0)
};

Particle pTemplate2 = {
    .m_rate: 10,  .m_speed: 5.f,  .m_duration: 5.f,  .m_looping: true,  .m_size: Vector2( x: .3f,  y: .3f),
    Shape::Sphere,  .m_texture: "",  .color: Vector3( x: 0,  y: 1,  z: 1)
};
pSys->addDefaultComponents();
m_pContext->getMeshManager()->registerAsset(pSHandle);

PE::Handle pParticleMeshInstance( dbgName: "MeshInstance2",  neededSize: sizeof(MeshInstance));
MeshInstance *pInstance = new(pParticleMeshInstance) MeshInstance( [&] *m_pContext, m_arena,  pParticleMeshInstance);
pInstance->addDefaultComponents();
pInstance->initFromRegisteredAsset( pSys);
RootSceneNode::Instance()->addComponent( pParticleMeshInstance);

pSys->createParticleSystem( pTemplate);
```

```
void ClientNetworkManager::debugRender(int &threadOwnershipMask, float xoffset /* = 0*/, float yoffset /* = 0*/)
{
    sprintf(PEString::s_buf,  _Format: "Client: %s Id: %d", EClientStateToString(m_state), m_clientId);
    DebugRenderer::Instance()->createTextMesh(
        PEString::s_buf,  isOverlay2D: false,  is3D: true,  is3DFacedToCamera: false,  is3DFacedToCameraLockedYAxis: false,  timeToLive: 0,
         pos: Vector3( x: xoffset,  y: yoffset,  z: 0),  scale: 1.0f,  [&] threadOwnershipMask);

    if (!getFirstComponent<RootSceneNode>())
    {
        addComponent( RootSceneNode::InstanceHandle());
    }
    ParticleSystem* pSys = RootSceneNode::Instance()->getFirstComponent<ParticleSystem>();
    pSys->loadParticle_needsRC( [&] threadOwnershipMask);
```