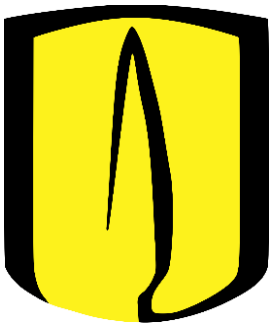
Computación Visual Interactiva



Proyecto Final

Estudiante:

Nicolas Miguel Murillo Cristancho

Objetivos del proyecto:

El presente proyecto busca crear un software de simulación de moléculas con fines educativos y académicos. En particular, se quiere usar Diligent graphics para crear un programa el cual pueda mostrar simulaciones de moléculas y sus interacciones de manera gráfica tanto a menor como a mayor escala. El programa deberá recibir como input información de moléculas y deberá estar preparado para recibir actualizaciones en tiempo real de las mismas, así como ejecutar acciones en tiempo real que simulen la interacción entre dos moléculas en tiempo real. Todos estos eventos serán modelados en 3D usando DiligentGraphics y mostrados al usuario en tiempo real. Las principales características deseables en el programa final son las siguientes: - Modelamiento de moléculas a nivel microscópico

-Modelamiento de reacciones (se desea mostrar, en función del tiempo, como cambian las moléculas poco a poco)

-Cantidad de moléculas (se espera poder procesar al menos cientos, y si es posibles, miles) - Tipos de moléculas (no todas las moléculas o todos los átomos existentes se podrán modelar)

Motivación

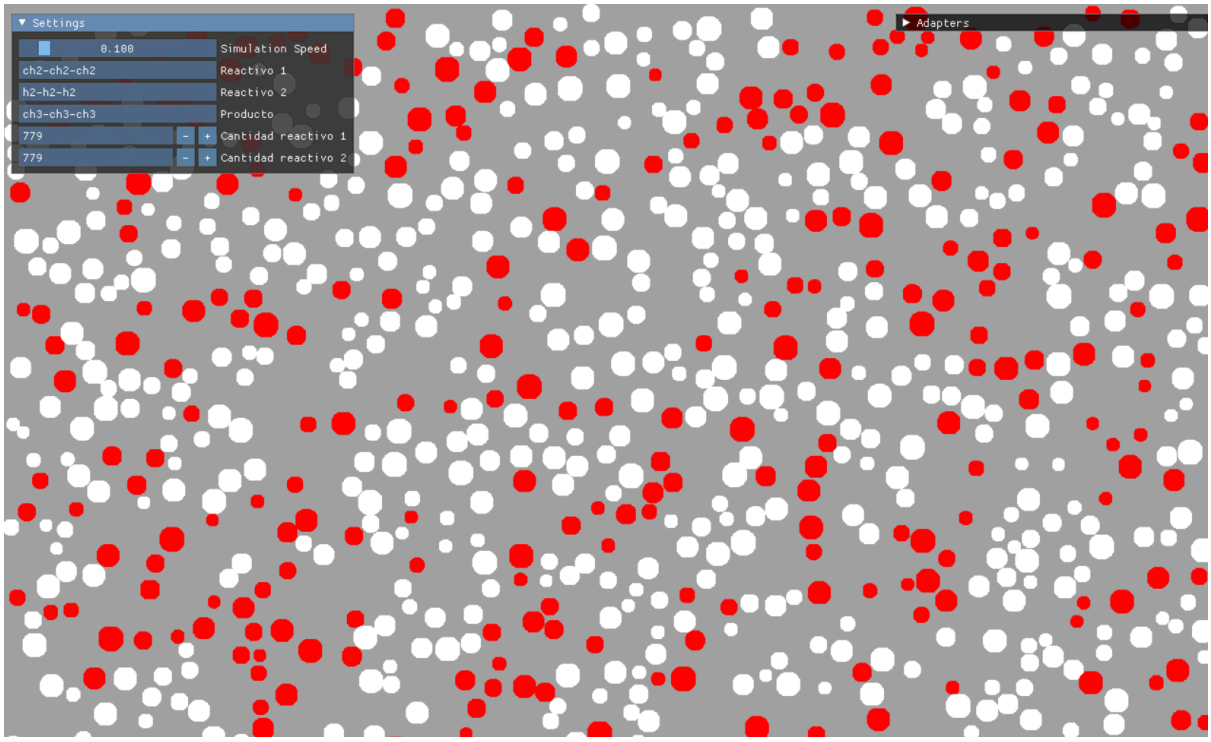
Crear un software educativo que ayude a muchos estudiantes de secundaria (por ejemplo, grado 11) a entender mejor el comportamiento de las moléculas, especialmente lo que respecta a química orgánica.

Crear un software interactivo en donde se puedan ver las diferencias en los procesos de reacciones moleculares tanto de manera detallada (pocas moléculas al tiempo) como de manera más ggeneralizada (muchas moléculas al tiempo)

Aportar con un simulador que ayude a comprender como se pueden romper y formar diferentes enlaces atómicos en la química orgánica

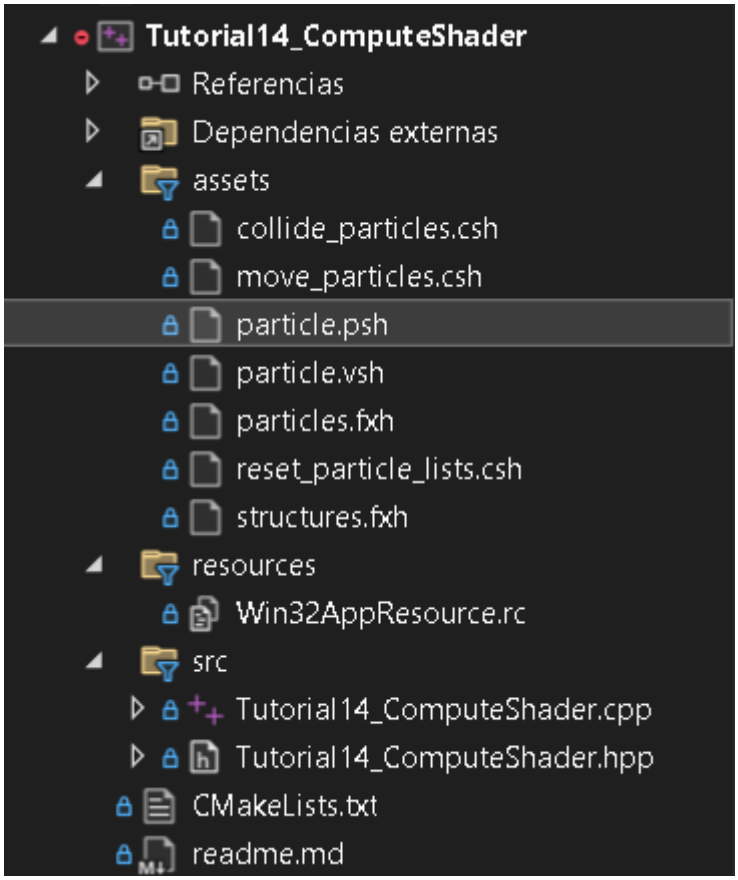
Aportar con controles de velocidad que permitan ver de manera acelerada o ralentizada los movimientos de las moléculas en una reacción determinada.

Capturas de la última versión del sistema:



Código base:

Se usó el código del tutorial 14 de dilligent graphics, el cuál cuenta con la siguiente estructura:



El archivo principal cuenta con varias funciones de las que destacan las siguientes:

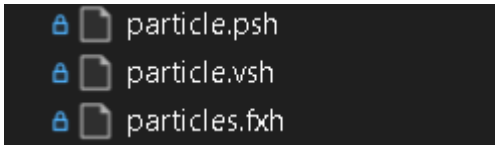
```cpp
void Tutorial14_ComputeShader::CreateParticleBuffers()
{
    m_pParticleAttribsBuffer.Release();
    m_pParticleListHeadsBuffer.Release();
    m_pParticleListsBuffer.Release();

    BufferDesc BuffDesc;
    BuffDesc.Name              = "Particle attribs buffer";
    BuffDesc.Usage             = USAGE_DEFAULT;
    BuffDesc.BindFlags         = BIND_SHADER_RESOURCE | BIND_UNORDERED_ACCESS;
    BuffDesc.Mode              = BUFFER_MODE_STRUCTURED;
    BuffDesc.ElementByteStride = sizeof(ParticleAttribs);
    BuffDesc.Size              = sizeof(ParticleAttribs) * m_NumParticles;

    std::vector<ParticleAttribs> ParticleData(m_NumParticles);

    std::mt19937 gen; // Standard mersenne_twister_engine. Use default seed
                      // to generate consistent distribution.

    std::uniform_real_distribution<float> pos_distr(-1.f, +1.f);
    std::uniform_real_distribution<float> size_distr(0.5f, 1.f);

    constexpr float fMaxParticleSize = 0.05f;
    float           fSize            = 0.7f / std::sqrt(static_cast<float>(m_NumParticles));
    fSize                            = std::min(fMaxParticleSize, fSize);
    for (ParticleAttribs& particle : ParticleData)
    {
        particle.f2NewPos.x   = pos_distr(gen);
        particle.f2NewPos.y   = pos_distr(gen);
        particle.f2NewSpeed.x = pos_distr(gen) * fSize * 5.f;
        particle.f2NewSpeed.y = pos_distr(gen) * fSize * 5.f;
        particle.fSize        = fSize * size_distr(gen);
    }

    BufferData VBData;
    VBData.pData    = ParticleData.data();
    VBData.DataSize = sizeof(ParticleAttribs) * static_cast<Uint32>(ParticleData.size());
    m_pDevice->CreateBuffer(BuffDesc, &VBData, &m_pParticleAttribsBuffer);
    IBufferView* pParticleAttribsBufferSRV = m_pParticleAttribsBuffer->GetDefaultView(BUFFER_VIEW_SHADER_RESOURCE);
    IBufferView* pParticleAttribsBufferUAV = m_pParticleAttribsBuffer->GetDefaultView(BUFFER_VIEW_UNORDERED_ACCESS);

    BuffDesc.ElementByteStride = sizeof(int);
    BuffDesc.Mode              = BUFFER_MODE_STRUCTURED;
    BuffDesc.Size              = Uint64{BuffDesc.ElementByteStride} * static_cast<Uint64>(m_NumParticles);
    BuffDesc.BindFlags         = BIND_UNORDERED_ACCESS | BIND_SHADER_RESOURCE;
    m_pDevice->CreateBuffer(BuffDesc, nullptr, &m_pParticleListHeadsBuffer);
    m_pDevice->CreateBuffer(BuffDesc, nullptr, &m_pParticleListsBuffer);
    IBufferView* pParticleListHeadsBufferUAV = m_pParticleListHeadsBuffer->GetDefaultView(BUFFER_VIEW_UNORDERED_ACCESS);
    IBufferView* pParticleListsBufferUAV     = m_pParticleListsBuffer->GetDefaultView(BUFFER_VIEW_UNORDERED_ACCESS);
    IBufferView* pParticleListHeadsBufferSRV = m_pParticleListHeadsBuffer->GetDefaultView(BUFFER_VIEW_SHADER_RESOURCE);
    IBufferView* pParticleListsBufferSRV     = m_pParticleListsBuffer->GetDefaultView(BUFFER_VIEW_SHADER_RESOURCE);
```

```cpp
void Tutorial14_ComputeShader::CreateRenderParticlePSO()
{
    GraphicsPipelineStateCreateInfo PSOCreateInfo;

    // Pipeline state name is used by the engine to report issues.
    PSOCreateInfo.PSODesc.Name = "Render particles PSO";

    // This is a graphics pipeline
    PSOCreateInfo.PSODesc.PipelineType = PIPELINE_TYPE_GRAPHICS;

    // clang-format off
    // This tutorial will render to a single render target
    PSOCreateInfo.GraphicsPipeline.NumRenderTargets             = 1;
    // Set render target format which is the format of the swap chain's color buffer
    PSOCreateInfo.GraphicsPipeline.RTVFormats[0]                = m_pSwapChain->GetDesc().ColorBufferFormat;
    // Set depth buffer format which is the format of the swap chain's back buffer
    PSOCreateInfo.GraphicsPipeline.DSVFormat                    = m_pSwapChain->GetDesc().DepthBufferFormat;
    // Primitive topology defines what kind of primitives will be rendered by this pipeline state
    PSOCreateInfo.GraphicsPipeline.PrimitiveTopology            = PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP;
    // Disable back face culling
    PSOCreateInfo.GraphicsPipeline.RasterizerDesc.CullMode      = CULL_MODE_NONE;
    // Disable depth testing
    PSOCreateInfo.GraphicsPipeline.DepthStencilDesc.DepthEnable = False;
    // clang-format on

    BlendStateDesc& BlendDesc = PSOCreateInfo.GraphicsPipeline.BlendDesc;

    BlendDesc.RenderTargets[0].BlendEnable = True;
    BlendDesc.RenderTargets[0].SrcBlend    = BLEND_FACTOR_SRC_ALPHA;
    BlendDesc.RenderTargets[0].DestBlend   = BLEND_FACTOR_INV_SRC_ALPHA;

    ShaderCreateInfo ShaderCI;
    // Tell the system that the shader source code is in HLSL.
    // For OpenGL, the engine will convert this into GLSL under the hood.
    ShaderCI.SourceLanguage = SHADER_SOURCE_LANGUAGE_HLSL;

    // OpenGL backend requires emulated combined HLSL texture samplers (g_Texture + g_Texture_sampler combination)
    ShaderCI.Desc.UseCombinedTextureSamplers = true;

    // Presentation engine always expects input in gamma space. Normally, pixel shader output is
    // converted from linear to gamma space by the GPU. However, some platforms (e.g. Android in GLES mode,
    // or Emscripten in WebGL mode) do not support gamma-correction. In this case the application
    // has to do the conversion manually.
    ShaderMacro Macros[] = {{"CONVERT_PS_OUTPUT_TO_GAMMA", m_ConvertPSOutputToGamma ? "1" : "0"}};
    ShaderCI.Macros      = {Macros, _countof(Macros)};

    // Create a shader source stream factory to load shaders from files.
    RefCntAutoPtr<IShaderSourceInputStreamFactory> pShaderSourceFactory;
    m_pEngineFactory->CreateDefaultShaderSourceStreamFactory(nullptr, &pShaderSourceFactory);
    ShaderCI.pShaderSourceStreamFactory = pShaderSourceFactory;
    // Create particle vertex shader
```

```cpp
void Tutorial14_ComputeShader::CreateUpdateParticlePSO()
{
    ShaderCreateInfo ShaderCI;
    // Tell the system that the shader source code is in HLSL.
    // For OpenGL, the engine will convert this into GLSL under the hood.
    ShaderCI.SourceLanguage = SHADER_SOURCE_LANGUAGE_HLSL;

    // OpenGL backend requires emulated combined HLSL texture samplers (g_Texture + g_Texture_sampler combination)
    ShaderCI.Desc.UseCombinedTextureSamplers = true;

    // Create a shader source stream factory to load shaders from files.
    RefCntAutoPtr<IShaderSourceInputStreamFactory> pShaderSourceFactory;
    m_pEngineFactory->CreateDefaultShaderSourceStreamFactory(nullptr, &pShaderSourceFactory);
    ShaderCI.pShaderSourceStreamFactory = pShaderSourceFactory;

    ShaderMacroHelper Macros;
    Macros.AddShaderMacro("THREAD_GROUP_SIZE", m_ThreadGroupSize);

    RefCntAutoPtr<IShader> pResetParticleListsCS;
    {
        ShaderCI.Desc.ShaderType = SHADER_TYPE_COMPUTE;
        ShaderCI.EntryPoint      = "main";
        ShaderCI.Desc.Name       = "Reset particle lists CS";
        ShaderCI.FilePath        = "reset_particle_lists.csh";
        ShaderCI.Macros          = Macros;
        m_pDevice->CreateShader(ShaderCI, &pResetParticleListsCS);
    }

    RefCntAutoPtr<IShader> pMoveParticlesCS;
    {
        ShaderCI.Desc.ShaderType = SHADER_TYPE_COMPUTE;
        ShaderCI.EntryPoint      = "main";
        ShaderCI.Desc.Name       = "Move particles CS";
        ShaderCI.FilePath        = "move_particles.csh";
        ShaderCI.Macros          = Macros;
        m_pDevice->CreateShader(ShaderCI, &pMoveParticlesCS);
    }

    RefCntAutoPtr<IShader> pCollideParticlesCS;
    {
        ShaderCI.Desc.ShaderType = SHADER_TYPE_COMPUTE;
        ShaderCI.EntryPoint      = "main";
        ShaderCI.Desc.Name       = "Collide particles CS";
        ShaderCI.FilePath        = "collide_particles.csh";
        ShaderCI.Macros          = Macros;
        m_pDevice->CreateShader(ShaderCI, &pCollideParticlesCS);
    }

    RefCntAutoPtr<IShader> pUpdatedSpeedCS;
    {
```

Código extendido:

Se agregó código en 3 archivos cuyas clases definen el comportamiento de los átomos:



Estas modificaciones permiten controlar

-La forma geométrica en como se representan

-El movimiento que hacen los mismos de acuerdo a la velocidad de reacción establecida

- Los colores de cada uno de los átomos de modo que se diferencien diferentes tipos de átomos (como por ejemplo, carbono rojo e hidrógeno blanco)

- Control del comportamiento de las colisiones.

Se agregó código también al archivo principal .cpp para los siguientes propósitos:

-Control de la cantidad de particulas iniciales en pantalla

- control de la cantidad de partículas de acuerdo a la ecuación del reactor 1

- control de la cantidad de partículas de acuerdo a la ecuación del reactor 2

- control de la cantidad de partículas de acuerdo al número de copias del reactor 1

- control de la cantidad de partículas de acuerdo al número de copias del reactor 2

- Control de la velocidad de reacción

- Campos de Imgui para escribir las fórmulas de los reactores.

Todas las anteriores modificaciones fueron las que permitieron la simulación de todo el proceso de reacción.

La arquitectura planteada en este caso consta del cpp, un archivo auxiliar de headers hpp y 7 archivos secundarios que definen los comportamientos de las partículas y demás variables de la simulación. En general, la arquitectura se divide en dos capas, siendo la inferior una capa que juega el papel de information holder para los objetos y la capa superior que juega el papel de ejecutor usando la información de la capa previa para reproducir la simulación.

En cuanto a los shaders, en este caso se usaron los compute shaders con 256 hilos paralelos.

Referencias:

Nicolas. (s. f.). *GitHub - nicolas23589/Proyecto-CVI*. GitHub. https://github.com/nicolas23589/CVI-Proy

DiligentGraphics. (s. f.). *GitHub - DiligentGraphics/DiligentEngine: A modern cross-platform low-level graphics library and rendering framework*. GitHub. https://github.com/DiligentGraphics/DiligentEngine