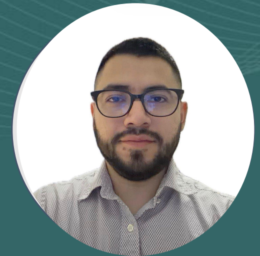
IWOCL 2026

FunGT: SYCL-Based Graphics Engine – Path Tracing, Particle Simulation, and Real-Time Interoperability on Discrete and Integrated GPUs.

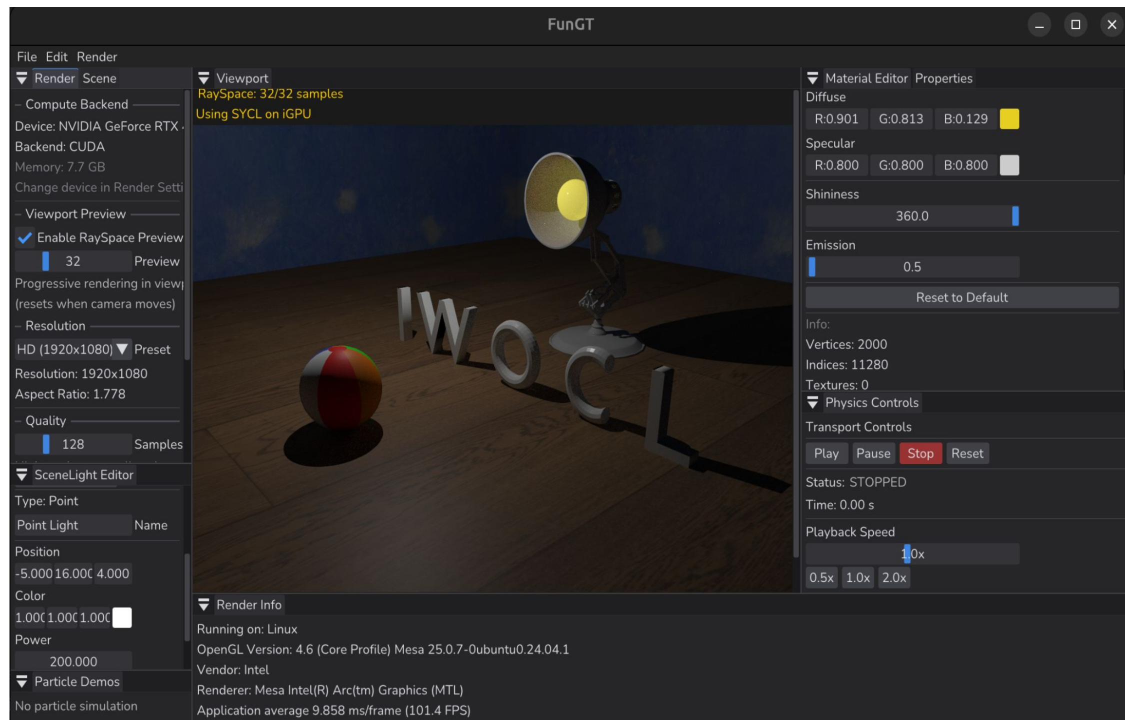
Juan Garcia



FunGT

A really Fun Graphics Tool

- Rendering options:
  - Rasterization using OpenGL
  - RaySpace: path tracer using SYCL
- Physics
  - Collisions
- Particle System



FunGT : How does it work ?

- DPC++ compiler built from source: https://github.com/intel/llvm
- NVCC (for pure cuda backend path tracer)

Example to enable both backends

cmake .. -DFUNGT_USE_SYCL=ON -DFUNGT_USE_CUDA=ON

Full instructions at: https://github.com/FunGTs/FunGT

FunGT

SYCL features:

- Bindless Images [1]
- Runtime Compilation [2]
- SYCL-OpenCL backend

Bindless Images and Runtime Compilation are very recent features

[1] Codeplay Software Ltd., “SYCL Bindless Images – An Introduction,” Feb. 11, 2025. [Online]. Available: https://codeplay.com/portal/blogs/2025/02/11/sycl-bindless-images

[2] Codeplay Software Ltd., “SYCL Runtime Compilation: A New Way to Specialise Kernels Using C++ Metaprogramming,” Jul. 8, 2025. [Online]. Available: https://codeplay.com/portal/blogs/2025/07/08/sycl-runtime-compilation

FunGT: Why OpenGL ?

FunGT was design in first place as an OpenGL "game engine"

OpenGL approach:

Use of **Compute Shaders** for Particle Systems: Simulating thousands of particles for effects.

SYCL-OpenCL-OpenGL : Why?

Use case:

“**Update thousands of particles using a SYCL kernel and display them on screen** ”

The interoperability between SYCL and OpenGL works, but only via OpenCL backend.

OpenCL-OpenGL interoperability is well known *[3]*.

[3] Scarpino, M. (2011). OpenCL in action: How to accelerate graphics and computations. Manning Publications

SYCL-OpenCL-OpenGL

```
// 1 Register a queue
flib::sycl_handler::register_queue("gl_queue",
                                   flib::device::GPU,
                                   flib::vendor::INTEL,
                                   flib::backend::OPENCL);
// 2 Create OpenGL interop (if needed)
flib::sycl_handler::create_gl_interop_context("gl_queue");
// 3 Print info (if needed)
flib::sycl_handler::get_device_info("gl_queue");
```

FunGT uses funlib an in-house library that handles queue creations

Full code: github.com/juanchuletas/funlib

SYCL-OpenCL-OpenGL

```
void flib::sycl_handler::create_gl_interop_context (const std::string& name)
{

   char extensions[2048];
   clGetDeviceInfo(clDev, CL_DEVICE_EXTENSIONS, sizeof(extensions), extensions,
nullptr);
   if(std::string(extensions).find("cl_khr_gl_sharing") == std::string::npos) {
       throw std::runtime_error("OpenCL device does not support OpenGL
interoperability.");
   }

   cl_context_properties props[] = {
           CL_GL_CONTEXT_KHR, (cl_context_properties)glxContext,
           CL_GLX_DISPLAY_KHR, (cl_context_properties)glxDisplay,
           CL_CONTEXT_PLATFORM, (cl_context_properties)clPlatform,
           0
   };

   cl_int err = CL_SUCCESS;
    clCtx = clCreateContext(props, 1, &clDev, nullptr, nullptr, &err);
   if (! clCtx || err != CL_SUCCESS)
       throw std::runtime_error("Failed to create OpenCL context for OpenGL
interoperability.");

   _syclCtx = sycl::make_context<sycl::backend::opencl>(_clCtx);


}
```

For OpenCL Context creation:

Searches for the ***cl_khr_gl_sharing*** extension and creates a sycl context using sycl::backend::opencl

SYCL-OpenCL-OpenGL : The pipeline

```
    // Get native handles
    sycl::queue Q = flib::sycl_handler::get_queue("gl_queue");
    cl_context clcontext = flib::sycl_handler::get_clContext();
    cl_command_queue clqueue = sycl::get_native<sycl::backend::opencl>(Q);

    // Wrap the GL buffer
    cl_mem clbuffer = clCreateFromGLBuffer(clcontext, CL_MEM_READ_WRITE, vbo, NULL);

    sycl::context syclCtx = flib::sycl_handler::get_sycl_context();
    // CRITICAL: Finish all pending OpenGL operations first
    glFinish();
    cl_event acquire_event;
    clEnqueueAcquireGLObjects(clqueue, 1, &clbuffer, 0, NULL, &acquire_event);
    clWaitForEvents(1, &acquire_event);  // Wait for acquisition to complete
    {
        //Wrap into SYCL and run kernel
       sycl::buffer<Particle<T>> buf =
                   sycl::make_buffer<sycl::backend::opencl, Particle<T>>(clbuffer, syclCtx);
       Q.submit([&](sycl::handler& cgh) {
               auto acc = buf.template get_access<sycl::access::mode::read_write>(cgh);

                /* SOME CRAZY KERNEL   */

        });

        Q.wait();
      }
     clFinish(clqueue);
     //Release OpenCL buffer
     cl_event release_event;
     clEnqueueReleaseGLObjects(clqueue, 1, &clbuffer, 0, NULL, &release_event);
     clWaitForEvents(1, &release_event);  // Wait for release to complete
     glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT | GL_BUFFER_UPDATE_BARRIER_BIT);
clReleaseMemObject(clbuffer);
```

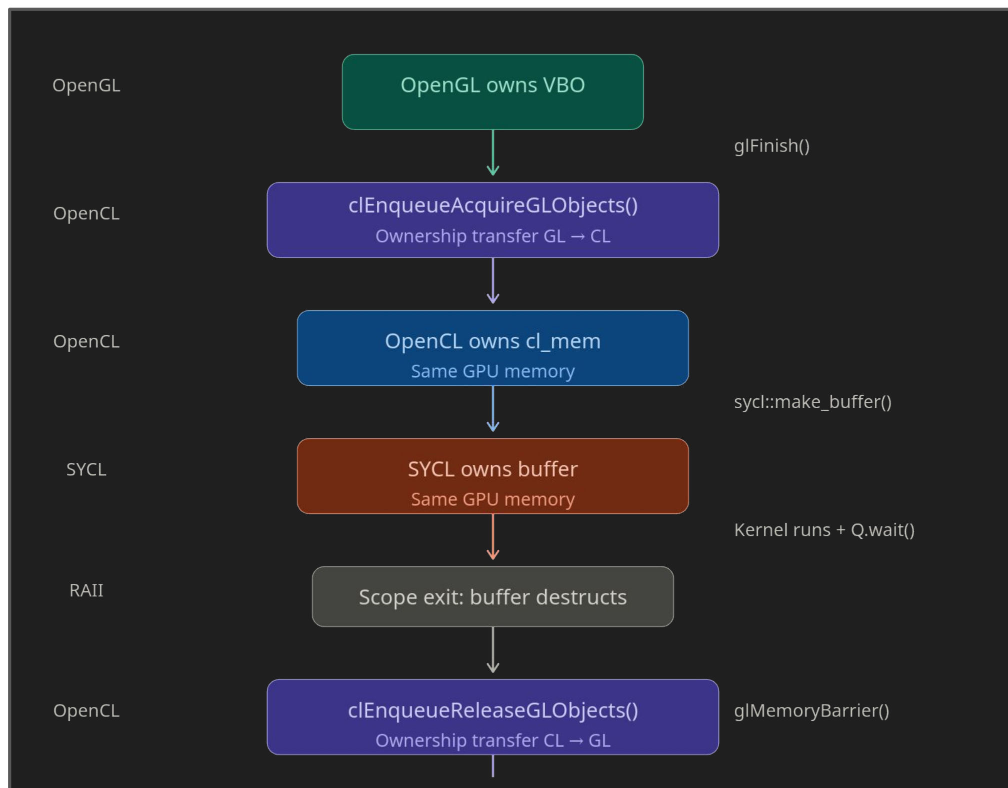
The code does three things:

1. Borrow the GL buffer for OpenCL-SYCL.

2. Run a kernel.

3. Give it back to GL

SYCL-OpenCL-OpenGL : The path



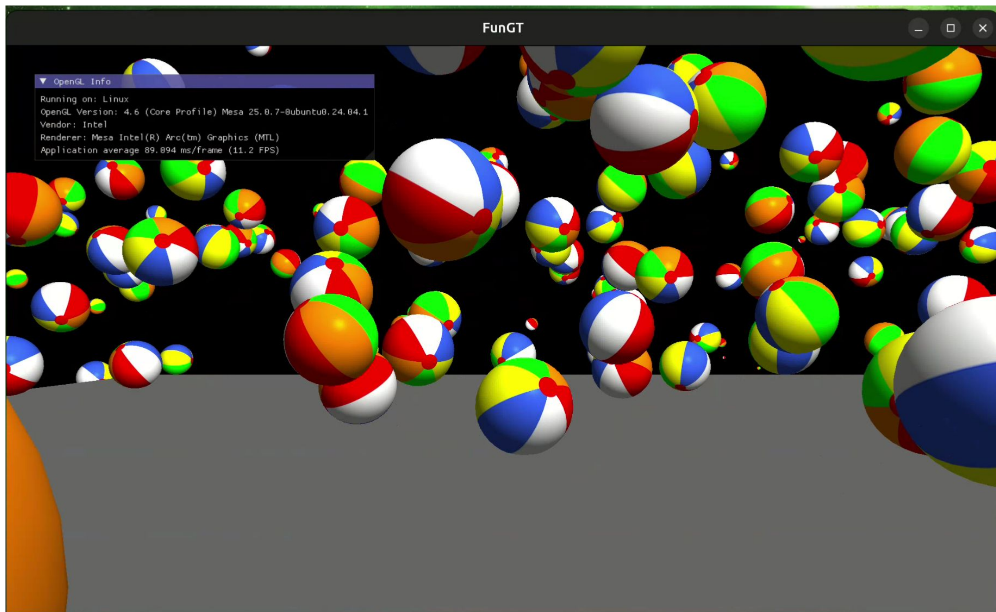
Never creates copies of the buffer. Same GPU memory, three different APIs taking turns owning it

SYCL-OpenCL-OpenGL : Usage

FunGT uses the interoperability in:

- Particle Simulation
- Physics: Collisions

SYCL-OpenCL-OpenGL : Physics



- Uses a Spatial Subdivision for the Broad-Phase
- Shader Storage Buffer Objects (SSBOs) to store matrices
- Positions are always on the GPU

Runtime Compilation RTC

FunGT has a kernel to update particles using lambdas

```
void static update(int numOfParticles, unsigned int vbo, Func
update_function, T dt){

Q.submit([&](sycl::handler& cgh) {
 /*SYCL-OpenCL-OpenGL interop code*/
auto acc = buf.template get access<sycl::access::mode::read write>(cgh);
cgh.parallel_for(sycl::range<2>(sycl::range<2>{static_cast<size_t>(xdim),

static cast<size t>(ydim)}),
   [=](sycl::item<2> item) {
       std::size t index = item[0] * ydim + item[1];
       if (index < n) {
       //user defined lambda function
       update_function(acc[index], dt);
       }
    });
});

}
```

Works using SYCL-OpenCL-OpenGL interoperability

Runtime Compilation RTC

1. Create the behavior

```
auto spiralExplosionUpdate = [](fgt::Particle<float>& p, float dt) {
    constexpr float central_force_strength = 0.5f;
    constexpr float spiral_speed = 0.5f;
    float distance = std::sqrt(p.position[0] * p.position[0]
                            + p.position[1] * p.position[1]);
    float radial_force = central_force_strength / (distance + 0.1f);
    p.velocity[0] += radial_force * p.position[0] * dt;
    p.velocity[1] += radial_force * p.position[1] * dt;
    p.velocity[0] -= spiral_speed * p.position[1] * dt;
    p.velocity[1] += spiral_speed * p.position[0] * dt;
    p.position[0] += p.velocity[0] * dt;
    p.position[1] += p.velocity[1] * dt;
    p.position[2] += p.velocity[2] * dt;
};
```

2. Use the update method:

```
fgt::ParticleSystem<float, decltype(fgt::spiralExplosionUpdate)>::update(
          numParticles, m_vbo.getId(), fgt::spiralExplosionUpdate, 0.005f);
```

Very verbose

Runtime Compilation RTC

This works great when the update function is known at compile time. But what if the user (or an IA) wants to write in the GUI at runtime?

Use case: The user types its own C++ code or request to the IA: “Show me a rain falling down from the sky”

Here is exactly where **sycl::kernel_compiler** is useful!

Runtime Compilation RTC

```
bool ParticleRTC::compileKernel(const std::string& user_code, std::string&
error_msg) {

std::string user_header = R"""(
        struct UserUpdate {
            void operator()(fgt::Particle<float>& p, float dt) const {
                )""" + user_code + R"""(
            }
        };
  )""";
 static constexpr auto sycl source = R"""(
       #include <sycl/sycl.hpp>
        #include "ParticleSimulation/particle.hpp"  // Physical file
        #include "Random/fgt rng.hpp"
        #include "user_update.h"                    // Virtual file

        extern "C" SYCL EXT ONEAPI FUNCTION PROPERTY((
            sycl::ext::oneapi::experimental::nd_range_kernel<2>))
        void particle_rtc_kernel(fgt::Particle<float>* particles, int n, int ydim,
float dt) {
            auto item = sycl::ext::oneapi::this work item::get_nd_item<2>();
            std::size t index = item.get_global_id(0) * ydim +
item.get_global_id(1);

            if (index < n) {
                UserUpdate update;
                update(particles[index], dt);
            }
        }
  )""";
```

- Receives the user code
- Stores the kernel to use it later

```
std::optional<sycl::kernel>
compiled_kernel =
exec_bundle.ext_oneapi_get_kernel(
"particle_rtc_kernel");
```

Runtime Compilation RTC

From the RTC blog:

```
SYCL_EXT_ONEAPI_FUNCTION_PROPERTY((
      sycl::ext::oneapi::experimental::nd_range_kernel<2>))
    void my_kernel(T *A, int B)
```

If we need to modify the parameter A, we must use a pointer:

```
float *A = malloc_shared<float>(N, q);
```

Runtime Compilation RTC

But FunGT uses SYCL-OpenCL-OpenGL for particle systems!

```
//Buffer
auto buf = sycl::make_buffer<sycl::backend::opencl,fgt::Particle<float>>(clbuffer,
syclCtx);
//Accessor
auto acc = buf.template get_access<sycl::access::mode::read_write>(cgh);
queue_.submit([&](sycl::handler& cgh) {

            cgh.set args(/* WHAT TO DO HERE * / )
             cgh.parallel_for(sycl::nd_range<2>{
                      ycl::range<2>{xdim, ydim},   // global size
                     sycl::range<2>{1, 1}          // local work-group size
            },
                Kernel_ref                        // Kernel
            );

});
```

First challenge!

Runtime Compilation RTC

There is no way to use Unified Share Memory under an OpenCL backend

```
   cl_mem clbuffer = clCreateFromGLBuffer(clcontext, CL_MEM_READ_WRITE, vbo,
NULL);

       queue_.submit([&](sycl::handler& cgh)  {

          // USE DIRECTLY THE clBuffer
           cgh.set_args(clbuffer,
                  static_cast<int>(n),
                  static_cast<int>(ydim), dt);
              cgh.parallel_for(
               sycl::nd_range<2>{
               sycl::range<2>{xdim, ydim},  // global size
                   sycl::range<2>{1, 1}          // local work-group size
           },
               kernel_ref
           );
       });
```

The Solution:

Use the clbuffer directly

Runtime Compilation RTC

The LLM doesn't generate SYCL code. It generates just the update body:

```
p.position[0] += p.velocity[0] * dt;
p.position[1] += p.velocity[1] * dt;
p.position[2] += p.velocity[2] * dt;
```

Runtime Compilation RTC

FunGT

Viewport

Particle RTC Editor

Describe the effect:

rain falling down, when particles go below y=-5 r
eset y position to 20 and keep falling,
no randomization on respawn

Generate from Prompt

Runtime Kernel Compiler

Status: Code generated! Review and compile.

Initialization Code:

Lambda: (Particle& p, int index)

```
fungt::RNG rng(index, 0);
p.position[0] = -10.0f + rng.nextFloat01() * 20.0f;
p.position[1] = 0.0f + rng.nextFloat01() * 20.0f;
p.position[2] = -10.0f + rng.nextFloat01() * 20.0f;
p.velocity[0] = 0.0f;
p.velocity[1] = -5.0f; // Constant downward velocity
```

Update Code:

Lambda: (Particle& p, float dt)

```
p.position[0] += p.velocity[0] * dt;
p.position[1] += p.velocity[1] * dt;
p.position[2] += p.velocity[2] * dt;

if (p.position[1] < -5.0f) {
    p.position[1] = 20.0f; // Reset y position to 20
```

Compile & Run

Physics Controls

Transport Controls

Play Pause Stop Reset

Particle RTC Editor

Describe the effect:

rain falling down, when particles go below y=-5 r
eset y position to 20 and keep falling,
no randomization on respawn

Runtime Compilation RTC

FunGT

Viewport

Particle RTC Editor

Describe the effect:

rain falling down, when particles go below y=-5 reset y position to 20 and keep falling, no randomization on respawn

Generate from Prompt

Runtime Kernel Compiler

Status: Code generated! Review and compile.

Initialization Code:

Lambda: (Particle& p, int index)

```
fungt::RNG rng(index, 0);
p.position[0] = -10.0f + rng.nextFloat01() * 20.0f;
p.position[1] = 0.0f + rng.nextFloat01() * 20.0f;
p.position[2] = -10.0f + rng.nextFloat01() * 20.0f;
p.velocity[0] = 0.0f;
p.velocity[1] = -5.0f; // Constant downward velocity
```

Update Code:

Lambda: (Particle& p, float dt)

```
p.position[0] += p.velocity[0] * dt;
p.position[1] += p.velocity[1] * dt;
p.position[2] += p.velocity[2] * dt;

if (p.position[1] < -5.0f) {
   p.position[1] = 20.0f; // Reset y position to 20
```

Compile & Run

Physics Controls

Transport Controls

Play Pause Stop Reset

Runtime Kernel Compiler

Status: Code generated! Review and compile.

Initialization Code:

Lambda: (Particle& p, int index)

```
fungt::RNG rng(index, 0);
p.position[0] = -10.0f + rng.nextFloat01() * 20.0f;
p.position[1] = 0.0f + rng.nextFloat01() * 20.0f;
p.position[2] = -10.0f + rng.nextFloat01() * 20.0f;
p.velocity[0] = 0.0f;
p.velocity[1] = -5.0f; // Constant downward velocity
```

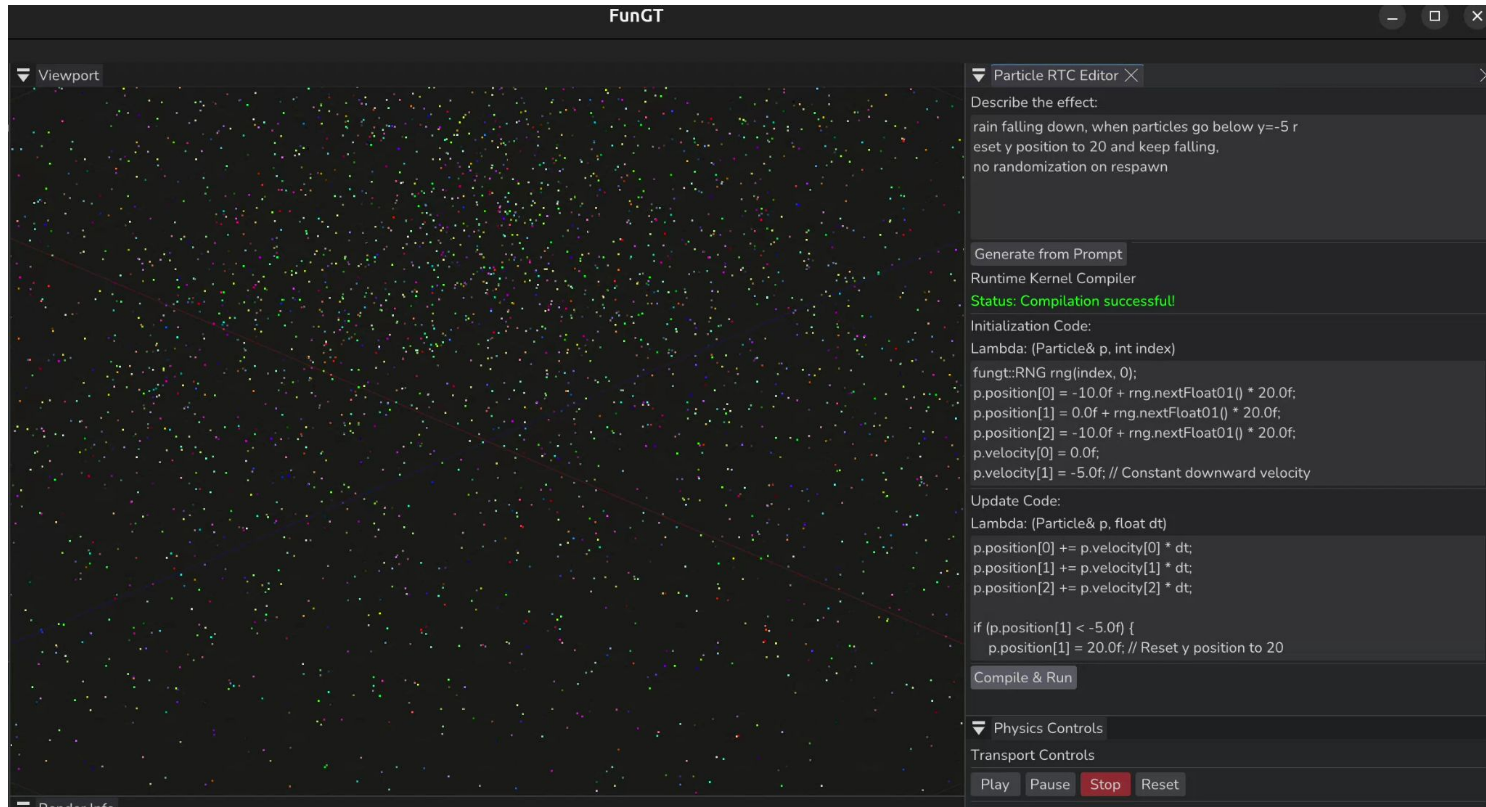
Update Code:

Lambda: (Particle& p, float dt)

```
p.position[0] += p.velocity[0] * dt;
p.position[1] += p.velocity[1] * dt;
p.position[2] += p.velocity[2] * dt;

if (p.position[1] < -5.0f) {
   p.position[1] = 20.0f; // Reset y position to 20
```

Runtime Compilation RTC



SYCL Bindless images : Path tracer

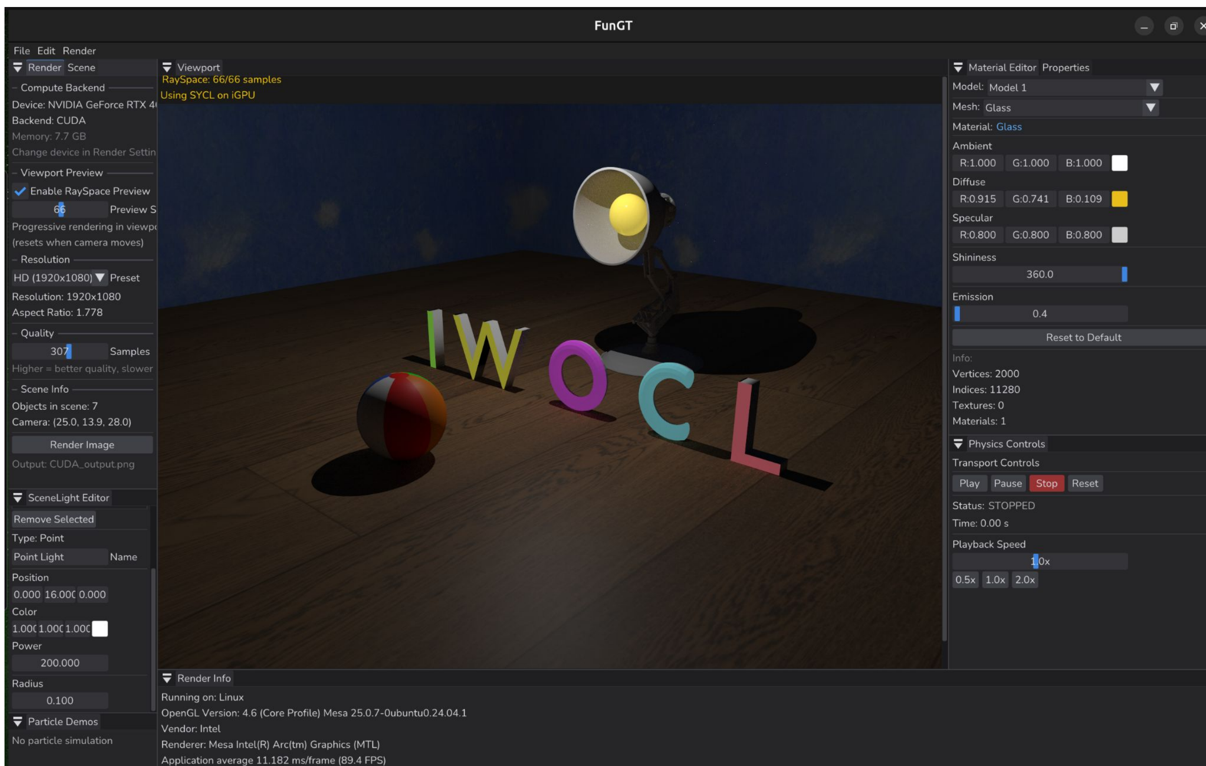
Render engine: RaySpace on Intel Arc iGPU with 32 default samples

RaySpace GPUs supported

- NVIDIA
  - CUDA
  - SYCL-CUDA backend
- Intel
  - SYCL

Path tracer viewport works with iGPU by default

SYCL Bindless images : Path tracer



Render engine: RaySpace on Intel Arc iGPU with 32 default samples

- Uses BVH (Bounding Volume Hierarchy)
- Emissive triangles support

SYCL Bindless images : Path tracer

```
class CUDA_Renderer : public IComputeRenderer{

   cudaTextureObject_t* m_textureObj= nullptr;
   int m_numTextures = 0;
     //More code of the class

}
```

CUDA: First version of the path tracer

- Uses the cudaTextureObject_t

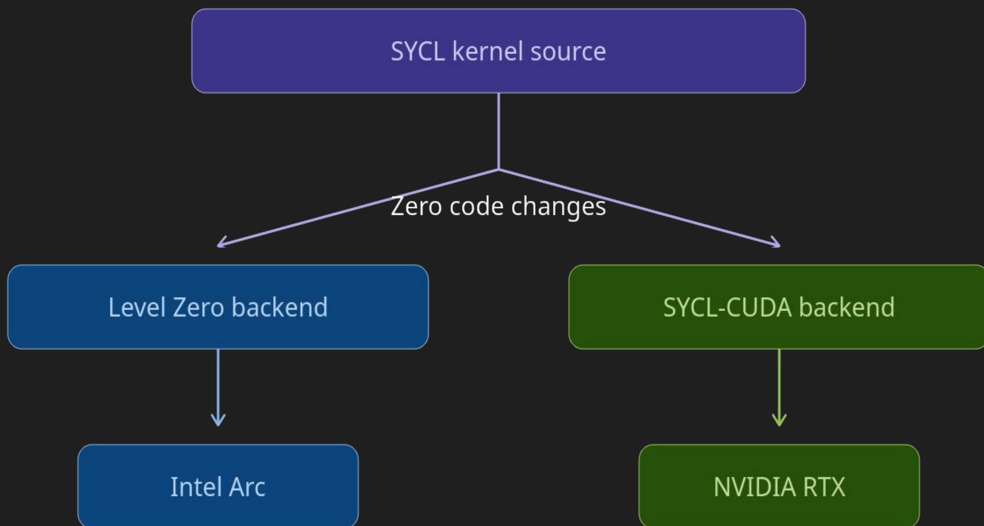
```
class SYCL_Renderer : public IComputeRenderer {
private:

sycl::ext::oneapi::experimental::sampled_image_handle*
m_textureHandles = nullptr;
   int m_numTextures = 0;
   sycl::queue m_queue;
```

SYCL:

- Uses the sycl bindless images

SYCL Bindless images : Path tracer



"Zero code changes between Intel and NVIDIA"

The same SYCL kernel with bindless images compiles and runs on both backends

SYCL Bindless images: What is missing?

sycl::image_channel_type has no sRGB. The hardware texture unit never applies gamma decode.

Manual decode in the kernel

```
texColor.x = powf(texColor.x, 2.2f);
texColor.y = powf(texColor.y, 2.2f);
texColor.z = powf(texColor.z, 2.2f);
```

FunGT: Hardware Support

Feature	Intel Arc	RTX (SYCL-CUDA)
GL interop	OpenCL backend	Not tested
Bindless images	Yes	Yes
kernel_compiler	Yes	Not tested

SYCL-CUDA-OpenGL backend is under development

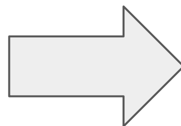
FunGT: Future Work

- SYCL-CUDA-OpenGL interop
  - Particle simulations and physics on NVIDIA via CUDA backend

FunGT: Future Work

- FunGT-dialect (MLIR IR) for particle systems
  - Safe code generation for LLM-generated GPU kernels
    https://github.com/FunGTs/fungt-dialect

```
p.velocity[2] += -9.8f * dt;
p.position[0] += p.velocity[0] * dt;
p.position[1] += p.velocity[1] * dt;
p.position[2] += p.velocity[2] * dt;
```



```
update rain:
    gravity = -9.8
    vel_z  =+ gravity * dt
    pos_x =+ vel_x * dt
    pos_y =+ vel_y * dt
    pos_z =+ vel_z * dt
```

FunGT-dialect

FunGT: Future Work

- ReSTIR (Reservoir-based Spatiotemporal Importance Resampling)
  - Temporal sample reuse for interactive path tracing preview

FunGT: Contact

Juan Garcia

Email: juan.garcia.cpp@gmail.com

FunGT repo: https://github.com/FunGTs/FunGT

Questions ?