Physics Engine: List of Improvements

1. Parallelization of Game Objects Update + Generate Collision Groups + Solve Collision Groups + Fill Render Data.
2. Game Data reorganization: fixed arrays vs std::vector.
3. Allow more game objects without a program's crash.
4. Fusion of Broad-Phase + Fine-Grained + Collision Groups.
5. Parallel merge sort instead of std::sort.
6. Instanced rendering: glDrawElementsInstanced + shaders rearrange.
7. Tiny changes: floats vs doubles, simple camera movement, constexpr, ...

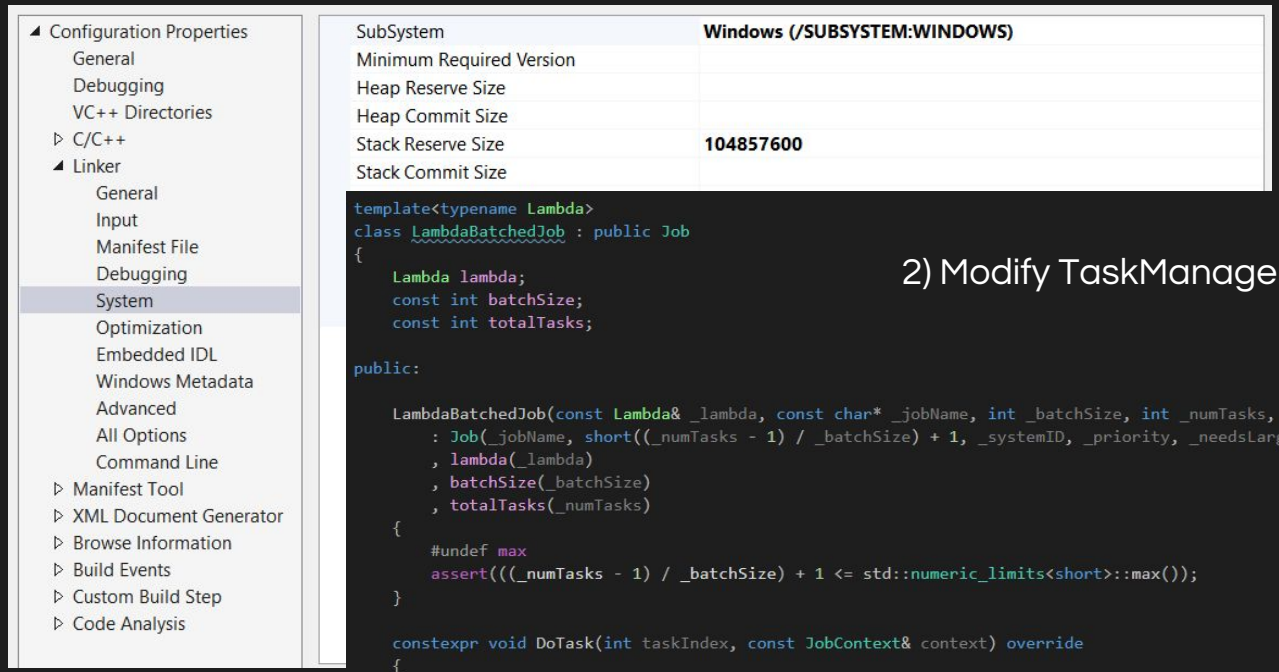
Parallelization of nearly everything

```
// UPDATE PHYSICS
{
    auto guard = context.CreateProfileMarkGuard("Update Physics");

    // 1 - Update posicions / velocitats
    UpdateGameObjects(gameData->gameObjects, renderData_, inputData, context);

    // 2 - Generació de colisions
    GenerateCollisionGroups(gameData, renderData_, context);

    // 3 - Resolució de colisions
    SolveCollisionGroups(gameData, context);
}

// FILL RENDER
FillRenderData(renderData_, gameData, context);
```

Game Data reorganization

```
static constexpr auto MaxGameObjects = 100'000u;
static constexpr auto GameObjectScale = 0.5f;

struct GameObjectList
{
    float posX[MaxGameObjects];
    float posY[MaxGameObjects];
    float velX[MaxGameObjects];
    float velY[MaxGameObjects];

    // Get extreme functions
    constexpr float getMinX(unsigned i) { return posX[i] - GameObjectScale; }
    constexpr float getMaxX(unsigned i) { return posX[i] + GameObjectScale; }
}
gameObjects;
```

Real parametrizable game objects size



1) Change stack capacity

2) Modify TaskManagersHelpers.h

```
template<typename Lambda>
class LambdaBatchedJob : public Job
{
    Lambda lambda;
    const int batchSize;
    const int totalTasks;

public:

    LambdaBatchedJob(const Lambda& _lambda, const char* _jobName, int _batchSize, int _numTasks, int _systemID = -1, Job::Priority _priority
        : Job(_jobName, short((_numTasks - 1) / _batchSize) + 1, _systemID, _priority, _needsLargeStack)
        , lambda(_lambda)
        , batchSize(_batchSize)
        , totalTasks(_numTasks)
    {
        #undef max
        assert(((_numTasks - 1) / _batchSize) + 1 <= std::numeric_limits<short>::max());
    }

    constexpr void DoTask(int taskIndex, const JobContext& context) override
    {
        int max = totalTasks < (taskIndex + 1) * batchSize ? totalTasks : (taskIndex + 1) * batchSize;
        for (int i = taskIndex * batchSize; i < max; ++i)
            LambdaCaller<Lambda, std::is_convertible<Lambda, std::function<void(int, const JobContext&)>>::value>(lambda, i, context);
    }
};
```

Fusion of collision groups generation

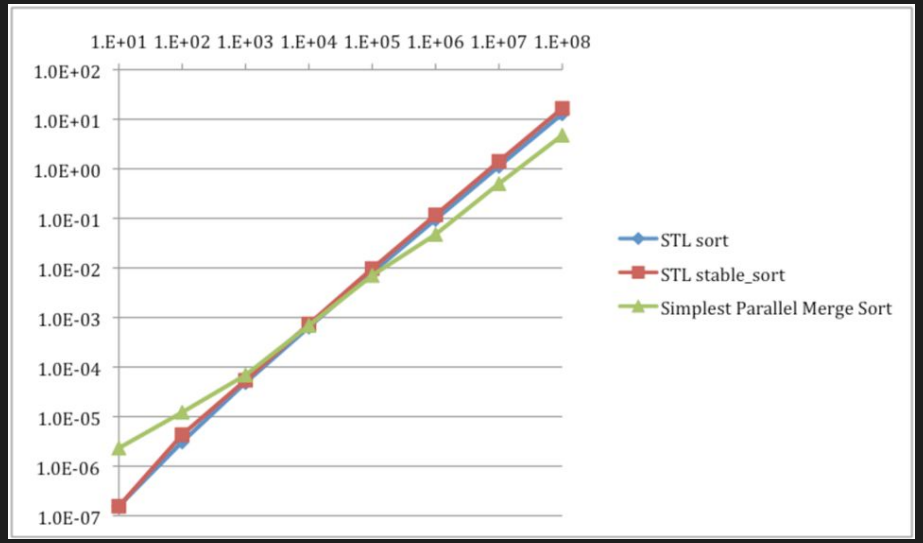
```
auto jobSFG = Utilities::TaskManager::CreateLambdaBatchedJob(
    [&gameData, &renderData, &createdGroups](int i, const Utilities::TaskManager::JobContext& context)
    {
        if (gameData->extremes[i].min)
        {
            for (int j = i + 1; j < GameData::ExtremesSize && gameData->extremes[i].index != gameData->extremes[j].index; ++j)
            {
                if (gameData->extremes[j].min && HasCollision(gameData->gameObjects, gameData->extremes[i].index, gameData->extremes[j].index))
                {
                    renderData.colors[gameData->extremes[i].index] = { 2, 0, 2, 1 };
                    renderData.colors[gameData->extremes[j].index] = { 0, 2, 2, 1 };
                    GenerateContactGroups(gameData,
                                          createdGroups,
                                          GenerateContactData(gameData->gameObjects, gameData->extremes[i].index, gameData->extremes[j].index),
                                          context);
                }
            }
        }
    },
    "Sweep + Fine-Grained + Collision Groups",
    GameData::ExtremesSize / (GameData::ExtremesSize < Utilities::Profiler::MaxNumThreads ? 1 : Utilities::Profiler::MaxNumThreads-1),
    GameData::ExtremesSize
);
context.DoAndWait(&jobSFG);
```

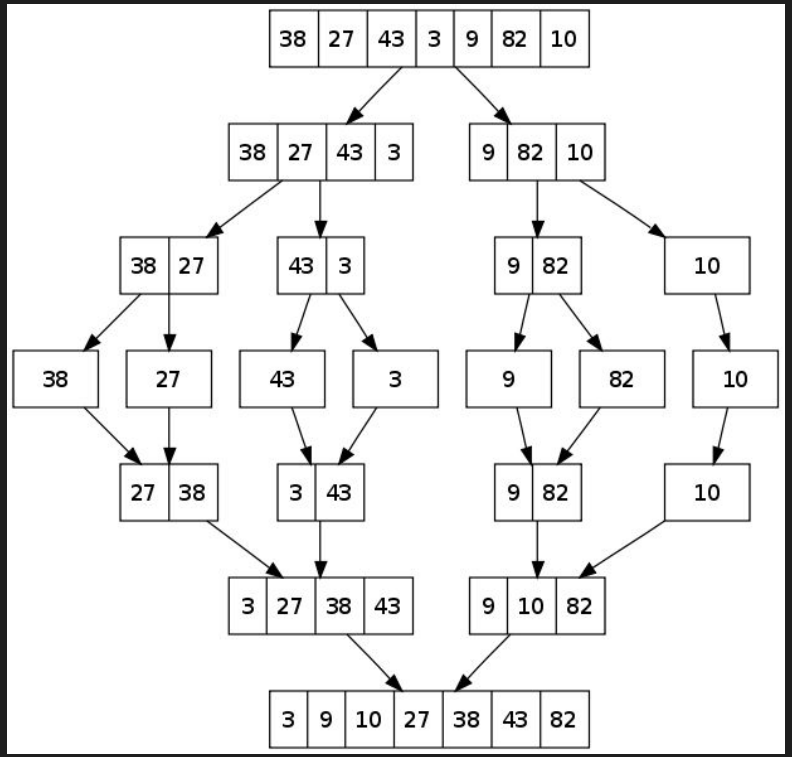
Parallel merge sort

std::sort -> Quick Sort (Intro Sort)

std::stable_sort -> Merge Sort

} **STL**



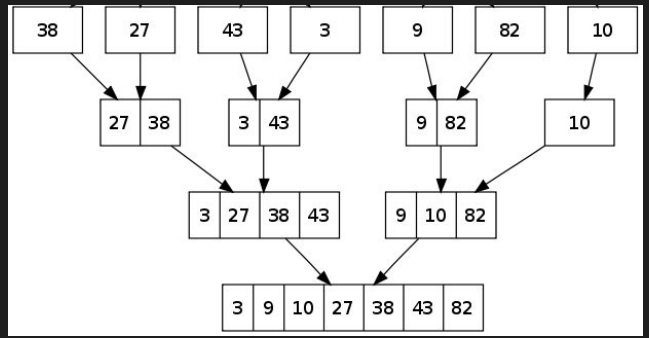


Parallel sort approach: Phase I


```
static constexpr auto NumMergeGroups = 8u; // NOTE: Only power of 2
std::atomic_int groupCounter = 0;
auto sortJob = Utilities::TaskManager::CreateLambdaJob(
    [&gameData, &groupCounter](int i, const Utilities::TaskManager::JobContext& context)
{
    const int index = groupCounter++;
    const int first = GameData::ExtremesSize * (float(index) / float(NumMergeGroups));
    const int last = GameData::ExtremesSize * (float(index + 1) / float(NumMergeGroups));
    std::sort(gameData->extremes + first, gameData->extremes + last);
},
    "Divide & Sort Extremes",
    NumMergeGroups
);
context.DoAndWait(&sortJob);
```

Parallel sort approach: Phase 2

```
while (mergeDivisions > 1)
{
    mergeDivisions /= 2;
    groupCounter = 0;
    std::copy(gameData->extremes, gameData->extremes + GameData::ExtremesSize, copiedExtremes);
    auto mergeJob = Utilities::TaskManager::CreateLambdaJob(
        [&gameData, &groupCounter, &mergeDivisions, &copiedExtremes](int i, const Utilities::TaskManager::JobContext& context)
    {
        const int index = groupCounter++;
        const int first = GameData::ExtremesSize * (float(index) / float(mergeDivisions));
        const int middle = first + float(GameData::ExtremesSize / mergeDivisions) / 2.0;
        const int last = GameData::ExtremesSize * (float(index + 1) / float(mergeDivisions));
        std::merge(copiedExtremes + first,
                   copiedExtremes + middle,
                   copiedExtremes + middle,
                   copiedExtremes + last,
                   gameData->extremes + first);
    },
        "Merge Extremes",
        mergeDivisions
        );
    context.DoAndWait(&mergeJob);
}
```



Instanced rendering: glDrawElements

```
void glDrawElements( GLenum mode,
                     GLsizei count,
                     GLenum type,
                     const GLvoid * indices);
```

```
void glDrawElementsInstanced( GLenum mode,
                              GLsizei count,
                              GLenum type,
                              const void * indices,
                              GLsizei primcount);
```

```
if (mode, count, or type is invalid )
    generate appropriate error
else {
    for (int i = 0; i < primcount ; i++) {
        instanceID = i;
        glDrawElements(mode, count, type, indices);
    }
    instanceID = 0;
}
```

Instanced rendering: Vertex shader

```
#version 420

layout (std140, binding = 0) uniform InstanceData // read from the buffer slot 0
{
    mat4 projection;
};

layout(location = 0) in vec3 in_Position;
layout(location = 1) in vec2 in_TexCoord;
layout(location = 2) in vec4 in_Color;
layout(location = 3) in mat4 in_Model;

out vec2 ex_TexCoord;
out vec4 ex_Color;

void main(void)
{
    gl_Position =  projection * in_Model * vec4(in_Position, 1.0);
    ex_TexCoord = vec2(in_TexCoord.s, 1.0 - in_TexCoord.t);
    ex_Color = in_Color;
}
```

```
struct RenderData
{
    enum class TextureID
    {
        BALL_WHITE,
        MAX_BALLS,
        PIXEL,
        DEARIMGUI,
        COUNT
    };

    TextureID texture = TextureID::BALL_WHITE;
    glm::mat4 modelMatrices[MaxGameObjects];
    glm::vec4 colors[MaxGameObjects];
};
```

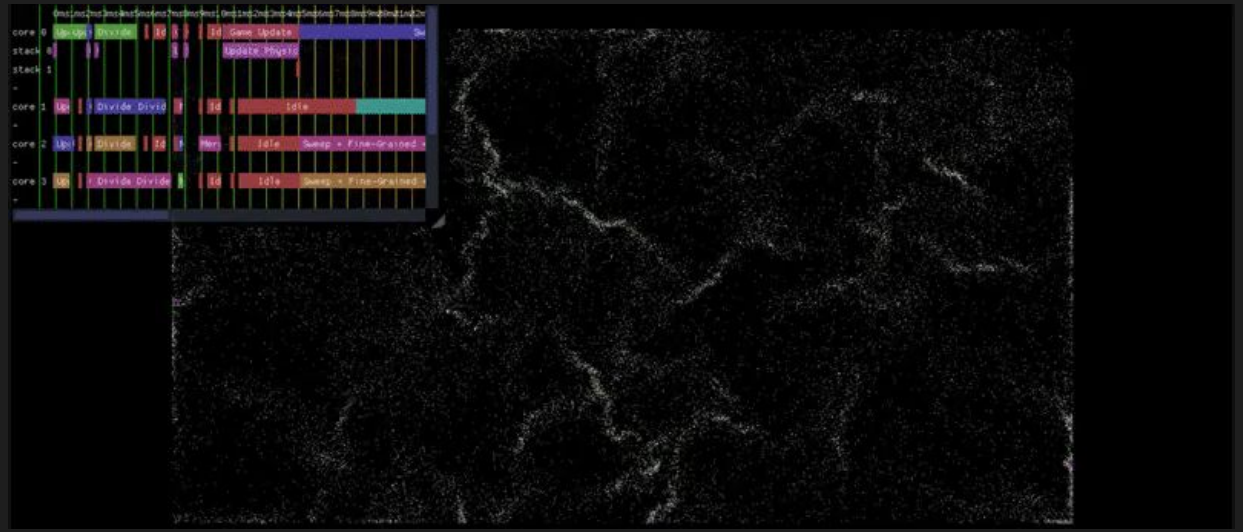
Instanced rendering: Draw in game loop

```
glBindTexture(GL_TEXTURE_2D, renderer.textures[static_cast<int>(Game::RenderData::TextureID::BALL_WHITE)]);

Win32::InstanceData instanceData{ projection };
glBindBuffer(GL_UNIFORM_BUFFER, renderer.uniforms[Win32::Renderer::GameScene]);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(Win32::InstanceData), static_cast<GLvoid*>(&instanceData));
glBindBufferBase(GL_UNIFORM_BUFFER, 0, renderer.uniforms[Win32::Renderer::GameScene]);

glBindBuffer(GL_ARRAY_BUFFER, renderer.sceneObjectData.vbo[Win32::SceneObjectData::VBO_InstanceModel]);
glBufferData(GL_ARRAY_BUFFER, Game::MaxGameObjects * sizeof glm::mat4, &renderData.modelMatrices[0], GL_DYNAMIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, renderer.sceneObjectData.vbo[Win32::SceneObjectData::VBO_Color]);
glBufferData(GL_ARRAY_BUFFER, Game::MaxGameObjects * sizeof glm::vec4, &renderData.colors[0], GL_DYNAMIC_DRAW);

glBindVertexArray(renderer.sceneObjectData.vao);
glDrawElementsInstanced(GL_TRIANGLES, renderer.sceneObjectData.numIndices,
                        GL_UNSIGNED_SHORT, nullptr, Game::MaxGameObjects);
glBindVertexArray(0);
```

Other tiny changes

```
projection = glm::ortho(-static_cast<float>((-camera.pos.x + inputData.windowHalfSize.x * camera.zoom)), // left
                        static_cast<float>((camera.pos.x + inputData.windowHalfSize.x * camera.zoom)), // right
                        -static_cast<float>((-camera.pos.y + inputData.windowHalfSize.y * camera.zoom)), // bot
                        static_cast<float>((camera.pos.y + inputData.windowHalfSize.y * camera.zoom)), // top
                        -5.0f, 5.0f); // near // far
```

```
enum {
    VBO_Vertex,
    VBO_Color,
    VBO_InstanceModel,
    VBO_MAX
};
GLuint vao;
GLuint vbo[VBO_MAX];
GLuint ebo;
int numIndices = 0;
```



Results

10k balls	~1 ms
50k balls	~11 ms
100k balls	~28 ms
150k balls	~51 ms
200k balls	~83 ms
500k balls	~290 ms
1M balls	~990 ms