```
"""
File: demolib/scene.cpp (231 bytes)
"""

#include "scene.h"

Scene::Scene()
{
    // TODO
}

void Scene::Render() const
{
    // TODO
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LEQUAL);
    glDepthRange(0.0f, 1.0f);

    //meshBuf.draw();
}


"""
File: demolib/program.cpp (17394 bytes)
"""

#include "demolib.h"

TextureBase::TextureBase() :
    allocated(false)
{
    glGenTextures(1, &id);
    LOG("Created texture %d", id);
}

TextureBase::~TextureBase()
{
    glDeleteTextures(1, &id);
}

GLuint TextureBase::currentId()
{
    if (!IS_DEBUG) INFO("WARNING: TextureBase::currentId called; slow");
    GLint curId;
    glGetIntegerv(GL_TEXTURE_BINDING_2D, &curId);
    return curId;
}

void Texture2DBase::bind(GLuint texUnit) const
{
    ASSERT(id != -1u, "Binding invalid texture");
    glActiveTexture(GL_TEXTURE0 + texUnit);
    glBindTexture(GL_TEXTURE_2D, id);
}

void Texture2DBase::unbind() const
{
    ASSERT(isInUse(), "Unbinding unbound texture %u", id);
    glBindTexture(GL_TEXTURE_2D, 0);
}

template <typename T>
void Texture2D<T>::allocate()
{
    if (allocated) {
        LOG("WARNING: Allocate on already allocated texture %u", id);
    }
    ASSERT(isInUse(), "Allocating unbound texture %u", id);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
    glTexImage2D(GL_TEXTURE_2D, 0, internalPixelFormat, w, h, 0, pixelFormat, pixelType, data);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
    allocated = true;
}

template <typename T>
void Texture2D<T>::update()
{
    ASSERT(allocated, "Updating unallocated texture %u", id);
    ASSERT(isInUse(), "Updating unbound texture %u", id);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, pixelFormat, pixelType, data);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
}

template <typename T>
void Texture2D<T>::updateLocal()
{
    ASSERT(allocated, "Locally updating unallocated texture %u", id);
    ASSERT(isInUse(), "Locally updating unbound texture %u", id);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, stride);
    glGetTexImage(GL_TEXTURE_2D, 0, pixelFormat, pixelType, data);
    glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
}

template <> const GLenum Texture2D<uint8_t>::pixelType = GL_UNSIGNED_BYTE;
template <> const GLenum Texture2D<float>::pixelType = GL_FLOAT;
template <> const GLenum Texture2D<glm::u8vec4>::pixelType = GL_UNSIGNED_BYTE;
template <> const GLenum Texture2D<uint8_t>::pixelFormat = GL_RED;
template <> const GLenum Texture2D<float>::pixelFormat = GL_RED;
template <> const GLenum Texture2D<glm::u8vec4>::pixelFormat = GL_RGBA;
template <> const GLenum Texture2D<uint8_t>::internalPixelFormat = GL_R8;
template <> const GLenum Texture2D<float>::internalPixelFormat = GL_R8;
template <> const GLenum Texture2D<glm::u8vec4>::internalPixelFormat = GL_RGBA8;

template struct Texture2D<uint8_t>;
template struct Texture2D<float>;
template struct Texture2D<glm::u8vec4>;

Shader::Shader(GLenum _type, const char *_name) :
    id(-1u),
    type(_type),
    name(_name)
{
    switch (type) {
        case GL_VERTEX_SHADER:   strType = "vertex"; break;
        case GL_GEOMETRY_SHADER: strType = "geometry"; break;
        case GL_FRAGMENT_SHADER: strType = "fragment"; break;
        default:                 strType = "unknown";
    }
}

std::shared_ptr<Shader> Shader::FromFile(GLenum type, const char *filename)
{
    std::shared_ptr<Shader> shader = std::make_shared<Shader>(type, filename);

    LOG("Opening %s shader program \"%s\"", shader->strType, filename);

    FILE *file = fopen(filename, "rb");
    ASSERT(file, "Failed to open %s shader program \"%s\"", shader->strType, filename);
    fseek(file, 0, SEEK_END);
    unsigned long fileLength = ftell(file);
    rewind(file);

    char *shaderCode = new char[fileLength];
    int shaderReadLength = fread(shaderCode, 1, fileLength, file);
    fclose(file);

    shader->code = std::string(shaderCode, shaderReadLength);
    delete[] shaderCode;

    return shader;
}

std::shared_ptr<Shader> Shader::Inline(GLenum type, std::string shaderCode, const char *name)
{
    std::shared_ptr<Shader> shader = std::make_shared<Shader>(type, name);
    shader->code = shaderCode;
    return shader;
}

bool Shader::compile()
{
    LOG("Compiling %s shader program \"%s\"", strType, name);

    id = glCreateShader(type);
    const char *codeStr = code.data();
    int codeLen = code.size();
    glShaderSource(id, 1, &codeStr, &codeLen);

    glCompileShader(id);

    GLint status;
    glGetShaderiv(id, GL_COMPILE_STATUS, &status);
    if (status == GL_FALSE) {
        GLint infoLogLength;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &infoLogLength);

        GLchar *strInfoLog = new GLchar[infoLogLength + 1];
        glGetShaderInfoLog(id, infoLogLength, NULL, strInfoLog);

        ASSERT(false, "Shader compile failure in %s shader \"%s\":\n%s",
               strType, name, strInfoLog);
        delete[] strInfoLog;
        return false;
    }

    return true;
}

Shader::~Shader()
{
    glDeleteShader(id);
}

constexpr U32 Program::PrimitiveRestartIndex;

Program::Program(std::initializer_list<std::shared_ptr<Shader> > &&_shaders) :
    id(-1u),
    shaders(_shaders)
{
}

void Program::use()
{
    ASSERT(currentId() == 0, "Program already in use");
    glUseProgram(id);
}

void Program::unuse()
{
    ASSERT(isInUse(), "Program not being used");
    glUseProgram(0);
}

GLuint Program::currentId()
{
    if (!IS_DEBUG) INFO("WARNING: Program::currentId called; slow");
    GLint curId;
    glGetIntegerv(GL_CURRENT_PROGRAM, &curId);
    return curId;
}

GLuint Program::doLink(std::vector<std::shared_ptr<Shader> > &shaders)
{
    for (auto &shader : shaders) {
        shader->compileLazy();
    }

    LOG("Linking shader program");
    GLuint id = glCreateProgram();
    for (auto const &shader : shaders) {
        glAttachShader(id, shader->id);
    }
    glLinkProgram(id);

    GLint status;
    glGetProgramiv(id, GL_LINK_STATUS, &status);
    if (status == GL_FALSE) {
        GLint infoLogLength;
        glGetProgramiv(id, GL_INFO_LOG_LENGTH, &infoLogLength);

        GLchar *strInfoLog = new GLchar[infoLogLength + 1];
        glGetProgramInfoLog(id, infoLogLength, NULL, strInfoLog);
        ASSERT(false, "Shader linker failure: %s\n", strInfoLog);
        delete[] strInfoLog;
        return -1u;
    }

    for (auto const &shader : shaders) {
        glDetachShader(id, shader->id);
    }

    return id;
}

bool Program::link()
{
    id = doLink(shaders);
    if (id == -1u) return false;

    postLink();
    return true;
}

GLint Program::uniformLocation(const char *name)
{
    GLint loc = glGetUniformLocation(id, name);
    if (loc == -1) {
        LOG("WARNING: no uniform named %s", name);
    }
    return loc;
}

bool Program::setUniform(const char *name, GLint s)
{
    GLint loc = uniformLocation(name);
    glUniform1i(loc, s);
    return loc != -1;
}

bool Program::setUniform(const char *name, float v)
{
    GLint loc = uniformLocation(name);
    glUniform1f(loc, v);
    return loc != -1;
}

bool Program::setUniform(const char *name, const glm::vec2 &v)
{
    GLint loc = uniformLocation(name);
    glUniform2fv(loc, 1, glm::value_ptr(v));
    return loc != -1;
}

bool Program::setUniform(const char *name, const glm::vec3 &v)
{
    GLint loc = uniformLocation(name);
    glUniform3fv(loc, 1, glm::value_ptr(v));
    return loc != -1;
}

bool Program::setUniform(const char *name, const glm::vec4 &v)
{
    GLint loc = uniformLocation(name);
    glUniform4fv(loc, 1, glm::value_ptr(v));
    return loc != -1;
}

bool Program::setUniform(const char *name, const glm::mat2 &v, bool trans)
```

```
    GLint loc = uniformLocation(name);
    glUniformMatrix2fv(uniformLocation(name), 1, trans, glm::value_ptr(v));
    return loc != -1;
}
bool Program::setUniform(const char *name, const glm::mat3 &v, bool trans)
{
    GLint loc = uniformLocation(name);
    glUniformMatrix3fv(uniformLocation(name), 1, trans, glm::value_ptr(v));
    return loc != -1;
}
bool Program::setUniform(const char *name, const glm::mat4 &v, bool trans)
{
    GLint loc = uniformLocation(name);
    glUniformMatrix4fv(uniformLocation(name), 1, trans, glm::value_ptr(v));
    return loc != -1;
}

const std::shared_ptr<Shader> ProgramMesh::vs =
    Shader::Inline(GL_VERTEX_SHADER, R"(
        #version 130

        in vec4 iPos;
        in vec2 iTexCoord;

        uniform mat4 uTransform;

        out vec2 vTexCoord;
        flat out float vVertexId;

        void main()
        {
            gl_Position = uTransform * iPos;
            vTexCoord = iTexCoord;
            vVertexId = gl_VertexID;
        }
    )", "ProgramMesh::vs");

const std::shared_ptr<Shader> ProgramMeshDebugVisFace::fs =
    Shader::Inline(GL_FRAGMENT_SHADER, R"(
        #version 130

        in vec2 vTexCoord;

        void main()
        {
            gl_FragColor = vec4(vTexCoord, 0.f, 1.f);
        }
    )", "ProgramMeshDebugVisFace::fs");

const std::shared_ptr<Shader> ProgramMeshDebugVisEdge::fs =
    Shader::Inline(GL_FRAGMENT_SHADER, R"(
        #version 130

        in vec2 vTexCoord;
        flat in float vVertexId;

        void main()
        {
            float edgePos = vVertexId*0.3f;
            gl_FragColor = vec4(edgePos-1.f, 1.f, edgePos, 1.f);
        }
    )", "ProgramMeshDebugVisEdge::fs");

void ProgramMesh::use()
{
    Program::use();
    glEnableClientState(GL_PRIMITIVE_RESTART_NV);
    glPrimitiveRestartIndexNV(PrimitiveRestartIndex);
}

void ProgramMesh::unuse()
{
    glDisableClientState(GL_PRIMITIVE_RESTART_NV);
    Program::unuse();
}

void ProgramMesh::doDraw(GLenum mode) const
{
    glBindVertexArray(vaoId);
    glDrawElements(mode, idxBuf.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}

void ProgramMesh::draw() const
{
    ASSERT(id != -1u, "Shader program not compiled");
    ASSERT(isInUse(), "Program not in use");

    doDraw(GL_TRIANGLE_FAN);
}

void ProgramMesh::drawWire() const
{
    ASSERT(id != -1u, "Shader program not compiled");
    ASSERT(isInUse(), "Program not in use");

    doDraw(GL_LINE_LOOP);
}

void ProgramMesh::updateMeshBuf(const Mesh &mesh, uint8_t vertMask, uint8_t faceMask)
{
    vertBuf.resize(mesh.verts.size());
    for (U32 vertIdx = 0; vertIdx < mesh.verts.size(); vertIdx++) {
        const MeshVert &vert = mesh.verts[vertIdx];
        if ((vert.mask | vertMask) != 0xFF) continue;
        vertBuf[vertIdx] = vert;
    }

    idxBuf.clear();
    for (U32 faceIdx = 0; faceIdx < mesh.faces.size(); faceIdx++) {
        const MeshFace &face = mesh.faces[faceIdx];
        if (face.material == 0xFF) continue;
        if ((face.mask | faceMask) != 0xFF) continue;
        if (!idxBuf.empty()) {
            idxBuf.push_back(PrimitiveRestartIndex);
        }
        for (U32 slotIdx = 0; slotIdx < face.count; slotIdx++) {
            idxBuf.push_back(face.verts[slotIdx]);
        }
    }

    glGenBuffers(1, &vertBufId);
    glGenBuffers(1, &idxBufId);
    glGenVertexArrays(1, &vaoId);
    glBindVertexArray(vaoId);

    glBindBuffer(GL_ARRAY_BUFFER, vertBufId);
    glBufferData(GL_ARRAY_BUFFER, vertBuf.size() * sizeof(vertBuf[0]),
                 vertBuf.data(), GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertBuf[0]), (GLvoid *) offsetof(VertBuf, pos));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(vertBuf[0]), (GLvoid *) offsetof(VertBuf, texCoord));

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, idxBufId);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, idxBuf.size() * sizeof(idxBuf[0]),
                 &idxBuf[0], GL_STATIC_DRAW);
    glBindVertexArray(0);
}

const std::shared_ptr<Shader> ProgramTexturedQuad::vs =
    Shader::Inline(GL_VERTEX_SHADER, R"(
        #version 130
        in vec4 iPos;
        uniform mat4 uTransform;
        out vec2 vTexCoord;

        void main()
        {
            gl_Position = uTransform*iPos;
            vTexCoord = iTexCoord;
        }
    )", "ProgramTexturedQuad::vs");
const std::shared_ptr<Shader> ProgramTexturedQuad::fs =
    Shader::Inline(GL_FRAGMENT_SHADER, R"(
        #version 130
        uniform sampler2D uTex;
        uniform bool uTexIsGray = false;

        void main()
        {
            vec4 color = texture(uTex, vTexCoord);
            if (uTexIsGray)
                color = color.rrra;
            gl_FragColor = color;
        }
    )", "ProgramTexturedQuad::fs");

ProgramTexturedQuad::ProgramTexturedQuad(GLuint _texUnit,
        std::initializer_list<std::shared_ptr<Shader> > &&_shaders) :
    ProgramMesh(std::move(_shaders)),
    texUnit(_texUnit)
{
    std::unique_ptr<Mesh> mesh = Mesh::createRing(4, PI/4.f, sqrt(2.f));
    updateMeshBuf(*mesh);
}

void ProgramTexturedQuad::postLink()
{
    ProgramMesh::postLink();
    use();
    setUniform("uTex", (GLint) texUnit);
    unuse();
}

const std::shared_ptr<Shader> ProgramTexQuadImp::vs =
    Shader::Inline(GL_VERTEX_SHADER, R"(
        #version 130
        uniform mat4 uTransform;
        uniform float uTexScale = 1.f;
        out vec2 vTexCoord;

        void main()
        {
            vec2 vPosition;
            switch (gl_VertexID)
            {
            case 0:
                // Bottom-left
                vPosition = vec2(-1.0, -1.0);
                break;
            case 1:
                // Top-left
                vPosition = vec2(-1.0, 1.0);
                break;
            case 2:
                // Bottom-right
                vPosition = vec2(1.0, -1.0);
                break;
            case 3:
                // Top-right
                vPosition = vec2(1.0, 1.0);
                break;
            }
            vTexCoord = vPosition*uTexScale;

            gl_Position = uTransform*vec4(vPosition, 0.f, 1.f);
            vTexCoord = 1.f;
            vTexCoord /= 2.f;
        }
    )", "ProgramTexQuadImp::vs");
const std::shared_ptr<Shader> ProgramTexQuadImp::fs =
    Shader::Inline(GL_FRAGMENT_SHADER, R"(
        #version 130
        uniform sampler2D uTex;
        uniform bool uTexIsGray = false;

        void main()
        {
            vec4 color = texture(uTex, vTexCoord);
            if (uTexIsGray)
                color = color.rrra;
            gl_FragColor = color;
        }
    )", "ProgramTexQuadImp::fs");

ProgramTexQuadImp::ProgramTexQuadImp(GLuint _texUnit,
        std::initializer_list<std::shared_ptr<Shader> > &&_shaders) :
    Program(std::move(_shaders)),
    texUnit(_texUnit)
{
}

void ProgramTexQuadImp::draw() const
{
    glBindVertexArray(vaoId);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindVertexArray(0);
}

void ProgramTexQuadImp::postLink()
{
    Program::postLink();
    use();
    setUniform("uTex", (GLint) texUnit);
    unuse();
}

FBO::FBO(int _w, int _h, int flags) :
    w(_w), h(_h), id(-1), depthBufId(-1)
{
    glGenFramebuffers(1, &id);
    glBindFramebuffer(GL_FRAMEBUFFER, id);
    if (flags & DEPTHBUF) {
        glGenRenderbuffers(1, &depthBufId);
        glBindRenderbuffer(GL_RENDERBUFFER, depthBufId);
        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                              w, h);
        glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                                  GL_DEPTH_ATTACHMENT,
                                  GL_RENDERBUFFER, depthBufId);
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
FBO::~FBO()
{
    glDeleteRenderbuffers(1, &depthBufId);
    glDeleteFramebuffers(1, &id);
}
void FBO::bindTexture(const Texture2DBase &tex, int attachment)
{
    ASSERT(tex.id != -1u, "Binding invalid texture");
    ASSERT(tex.allocated, "Binding unallocated texture %u", tex.id);
    ASSERT(w == tex.w && h == tex.h, "Binding texture of incorrect size to FBO"
           "(FBO %dx%d, tex %dx%d)",
           w, h, tex.w, tex.h);
    glBindFramebuffer(GL_FRAMEBUFFER, id);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0+attachment,
                           GL_TEXTURE_2D, tex.id, 0);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
#define CASE(x) case x: return #x
static const char *frameBufferError(GLenum err) {
    switch (err) {
        CASE(GL_FRAMEBUFFER_COMPLETE);
        CASE(GL_FRAMEBUFFER_UNDEFINED);
        CASE(GL_FRAMEBUFFER_UNSUPPORTED);
        CASE(GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT);
        CASE(GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT);
```

```
        CASE(GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER);
        CASE(GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER);
        CASE(GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE);
        CASE(GL_FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS);
        case 0: return "<error>";
        default: return "<unknown>";
    }
}
#undef CASE

void GLContext::bindFBO(FBO &fbo)
{
    if (&fbo == curFBO) return;
    if (curFBO) unbindFBO();
    glBindFramebuffer(GL_FRAMEBUFFER, fbo.id);
    ASSERT(glCheckFramebufferStatus(GL_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE,
           "Bind an incomplete framebuffer (error %s)",
           FrameBufferError(glCheckFramebufferStatus(GL_FRAMEBUFFER)));
    glPushAttrib(GL_VIEWPORT_BIT);
    glViewport(0, 0, fbo.w, fbo.h);
    curFBO = &fbo;
}
void GLContext::unbindFBO()
{
    if (!curFBO) {
        LOG("WARNING: unbinding FBO when no FBO attached; ignoring");
        return;
    }
    glPopAttrib();
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    curFBO = nullptr;
}


"""
File: demolib/text.cpp (5591 bytes)
"""

#include "text.h"
#include "SDL_ttf.h"
#include <string>

Font::Font(std::string file, int size)
{
    font = TTF_OpenFont(file.c_str(), size);
    ASSERT(font, "Font %s @ %dpt failed to load", file.c_str(), size);
}

Font::~Font()
{
    TTF_CloseFont(font);
}

glm::i32vec2 Font::measure(std::string text)
{
    glm::i32vec2 size;
    if (!TTF_SizeText(font, text.c_str(), &size.x, &size.y)) {
        ASSERT(false, "Font: could not measure size");
    }
    return size;
}

SDL_Surface *Font::draw(std::string text)
{
    SDL_Surface *ret = TTF_RenderText_Solid(font, text.c_str(), SDL_Color{0,0,0,0});
    ASSERT(ret, "Font: could not draw text");
    return ret;
}

TextureTextSDF::TextureTextSDF(int w, int h, SDL_Surface *surf, int spread) :
    Texture2D<uint8_t>(w, h, new uint8_t[w*h])
{
    uint8_t *sdata = static_cast<uint8_t *>(surf->pixels);
    float xscale = (float) surf->w / w,
          yscale = (float) surf->h / h;
    LOG("Text SDF: scales are %fx%f", (double) xscale, (double) yscale);

    /* twiddle the input surface
     * currently bit 1 (lsb) indicates the inside of the font
     * this loop makes bit 2 indicate the border of the font */
    for (int y = 0; y < surf->h; ++y) {
        for (int x = 0; x < surf->w; ++x) {
            if (sdata[y*surf->pitch + x] == 0) {
                continue;
            }
            const int dx[8] = {-1, 0, 1, 1, 1, 0, -1,-1},
                      dy[8] = {-1,-1, 1, 1, 1, 1, 1, 0};
            for (int i = 0; i < 8; ++i) {
                int xy = y + dy[i],
                    xx = x + dx[i];
                if (xy >= 0 && xy < surf->h &&
                    xx >= 0 && xx < surf->w) {
                    if (sdata[xy*surf->pitch + xx] == 0) {
                        sdata[y*surf->pitch + x] |= 0x02;
                        break;
                    }
                }
            }
        }
    }

    /* ty, tx are the texture pixel positions
     * ty*yscale, tx*xscale are the corresponding source pixel positions */
    for (int ty = 0; ty < h; ++ty) {
        float tyScaled = (ty+0.5f) * yscale;
        ASSERT(tyScaled >= 0 && tyScaled < surf->h, "got invalid source y %d", tyScaled);
        int prevSize = 0;
        for (int tx = 0; tx < w; ++tx) {
            int txScaled = (tx+0.5f) * xscale;
            ASSERT(txScaled >= 0 && txScaled < surf->w, "got invalid source x %d", txScaled);
            /* optimization: lower bound in xscale less than to left */
            int minSize = prevSize - xscale;
            /* upper bound on how far away the edge is */
            int maxSize = spread;
            for (int rSize = minSize; rSize <= maxSize; ++rSize) {
                int rl = txScaled-rSize,  rl = std::max(rl, 0),
                    tr = tyScaled-rSize, brr = std::min(rr, surf->w-1),
                    rb = tyScaled-rSize, brr = std::min(rb, 0),
                    rt = tyScaled+rSize, brt = std::min(rt, surf->h-1);
                auto check = [&] (int xx, int yy) {
                    ASSERT(xx >= 0 && xx < surf->w &&
                           yy >= 0 && yy < surf->h, "got invalid source pixel %d %d", xx, xy);
                    if ((sdata[yy*surf->pitch + xx] & 0x02) == 0) {
                        prevSize = rSize;
                        /* now we know our source pixel is an edge */
                        /* this is a new upper bound on how far away the edge is */
                        int newMaxSize = hypot(ty - tyScaled, xx - txScaled);
                        maxSize = std::min(maxSize, newMaxSize);
                    }
                };
                if (rl >= 0)       for (int xx = brl, xy = brb; xy <= brt; ++xy) check(xx, xy);
                if (rr < surf->w)  for (int xx = brl, xy = brb; xy <= brt; ++xy) check(xx, xy);
                if (rb >= 0)       for (int xx = brl, xy = brt; xx <= brr; ++xx) check(xx, xy);
                if (rt < surf->h)  for (int xx = brl, xy = brt; xx <= brr; ++xx) check(xx, xy);
                data[ty*w + tx] = tx;
            }
            /* now maxSize stores how far away an edge is */
            /* calculate the texel value */
            int sign = (sdata[tyScaled*surf->pitch + txScaled] & 1) ? 1 : -1;
            data[ty*w + tx] = 128 + sign * (maxSize*127./spread);
            prevSize = maxSize;
        }
    }
    SDL_FreeSurface(surf);
}

const std::shared_ptr<Shader> ProgramTextSDF::fs =
    Shader::Inline(GL_FRAGMENT_SHADER, R"(
        #version 130
        in vec2 vTexCoord;
        uniform sampler2D uTex;

        struct TextSDFParms {
            vec4 baseColor;
            bool glow;
            vec4 glowColor;
            float threshold;
        };
        uniform TextSDFParms uTextParms = TextSDFParms{
            vec4(1.f, 1.f, 1.f, 1.f),
            false,
            vec4(0.f, 1.f, 0.f, 1.f),
            0.5
        };
        void main()
        {
            vec4 color = uTextParms.baseColor;
            float texv = texture(uTex, vTexCoord).r;
            color.a = float(texv >= uTextParms.threshold);
            if (uTextParms.glow) {
                vec4 glowColor = uTextParms.glowColor */
                    smoothstep(uTextParms.threshold*0.5f,
                               uTextParms.threshold*1.0f, texv);
                color = mix(glowColor, color, color.a);
            }
            gl_FragColor = color;
        }
    )", "ProgramTextSDF::fs");

ProgramTextSDF::ProgramTextSDF(GLuint texUnit,
        std::initializer_list<std::shared_ptr<Shader> > &&_shaders) :
    ProgramTexQuadImp(texUnit, std::move(_shaders))
{
}


"""
File: demolib/mesh.cpp (20245 bytes)
"""

#include "mesh.h"
#include "demolib.h"
#include <cmath>
#include <cstring>    // memmove
#include <map>

/* Mesh::check checks a mesh for contradictions and constraint violations:
 *
 * for every half-edge e:
 *   * e = opposite(opposite(e)) == opposite(e)
 *   * start(e) = start(next(opposite(e)))
 *   * all faces are planar and convex with vertices anticlockwise
 */
bool Mesh::check() const
{
    for (U32 faceIdx = 0, faceSize = faces.size(); faceIdx < faceSize; faceIdx++) {
        ASSERT(faces[faceIdx].count >= 2 &&
               faces[faceIdx].count < faceSize,
               "Mesh::check failed: invalid number of slots (%u) on face %u",
               faces[faceIdx].count, faceIdx);
        for (U32 slot = 0; slot < faces[faceIdx].count; slot++) {
            U32 edge = edge(faceIdx, slot);
            U32 oppEdge = oppposite(edge);
            ASSERT(eCheck(oppEdge),
                   "Mesh::check failed: invalid oppEdge(e) for "
                   "(face %u, slot %u)", faceIdx, slot);
            U32 oppOEdge = oppposite(oppEdge);
            if (edge != oppOEdge) {
                LOG("Mesh::check failed: e != opposite(opposite(e)) for "
                    "(face %u, slot %u)", faceIdx, slot);
                return false;
            }
            if (edge == oppEdge) {
                LOG("Mesh::check failed: e == opposite(e) for "
                    "(face %u, slot %u)", faceIdx, slot);
                return false;
            }
            if (!samePosVert(vert2Idx(edge), vert2Idx(next(oppEdge)))) {
                LOG("Mesh::check failed: not similar to vert(next(oppEdge(e))) for "
                    "(face %u, slot %u)", faceIdx, slot);
                return false;
            }
        }
    }
    // TODO
    return true;
}

bool Mesh::checkFlags() const
{
    for (unsigned i = 0, vertSize = verts.size(); i < vertSize; i++) {
        if (verts[i].mask) return false;
    }
    for (unsigned i = 0, faceSize = faces.size(); i < faceSize; i++) {
        if (faces[i].mask) return false;
    }
    return true;
}

void Mesh::debugOut() const
{
    for (unsigned f = 0; f < faces.size(); f++) {
        for (unsigned s = 0; s < faces[f].count; s++) {
            U32 opp = edge(f,s);
            const glm::vec3 &pos = verts[vertIdx(f,s)].pos;
            const glm::vec2 &texCoord = verts[vertIdx(f,s)].texCoord;
            LOG("(%u,%u) is v%u. Opposite: (%u,%u) Pos: (%2.3f,%2.3f,%2.3f), Tex: (%2.3f,%2.3f)",
                f, s, vertIdx(f,s), opp/8, opp%8, (double) pos.x, (double) pos.y, (double) pos.z, (double) texCoord.x, (double) texCoord.y);
        }
    }
}

std::unique_ptr<Mesh> Mesh::createRing(unsigned sides, bool crease, float phase, float radius)
{
    ASSERT(sides >= 1 && sides <= MAXVERT,
           "Mesh::createRing with invalid number of vertices (%u)", sides);

    std::unique_ptr<Mesh> m{new Mesh};
    m->verts.resize(sides * (crease ? 2 : 1));

    m->faces.resize(2);
    m->faces[0].count = sides;
    m->faces[1].count = sides;
    m->faces[0].material = 0;
    m->faces[1].material = 0;

    float curRot = 0.f, incRot = 2*PI / sides;
    for (unsigned slot = 0; slot < sides; slot++, curRot += incRot) {
        MeshVert &thisVert = m->verts[slot];
        glm::vec3 circlePos = glm::vec3{cos(phase+curRot)*radius,
                                        sin(phase+curRot)*radius,
                                        0.f};
        thisVert.texCoord = glm::vec2{thisVert.pos} + glm::vec2{1.f,1.f} / 2.f;

        if (crease) {
            thisVert.normal = glm::vec3{0.f, 0.f, 1.f};

            MeshVert &otherVert = m->verts[sides + slot];
            otherVert.pos = thisVert.pos;
            otherVert.normal = glm::vec3{0.f, 0.f, -1.f};
            otherVert.texCoord = thisVert.texCoord;

            m->faces[0].verts[slot] = slot;
            m->faces[1].verts[slot] = sides + slot;
        } else {
            thisVert.normal = thisVert.pos;

            m->faces[0].verts[slot] = slot;
            m->faces[1].verts[slot] = (sides-slot)%sides;
        }

        m->faces[0].opposite[slot] = m->edge(1, sides-slot-1);
        m->faces[1].opposite[sides-slot-1] = m->edge(0, slot);
    }
}
```

```cpp
    ASSERT(m->check());

    return m;
}

std::unique_ptr<Mesh> Mesh::createGrid(int w, int h)
{
    std::unique_ptr<Mesh> m = Mesh::createRing(4, true, 5*PI/4.f, sqrt(2));

    /* first slice vertically from left to right */
    const glm::vec3 botL = m->verts[m->vertIdx(0,0)].pos,
                    botR = m->verts[m->vertIdx(0,1)].pos,
                    topL = m->verts[m->vertIdx(0,3)].pos,
                    topR = m->verts[m->vertIdx(0,2)].pos;

    for (int x = 0; x < w-1; ++x) {
        LOG("%d", x);
        U32 mTopNew = m->splitVert(m->eEdge(x*2,0), m->eVertPrev(m->eEdge(x*2,0)));
        m->verts[m->vertIdx(mBotNew)].pos = glm::mix(botL, botR, static_cast<float>(x+1)/w);
        m->verts[m->vertIdx(m->eVertPrev(mBotNew))].pos = glm::mix(botL, botR, static_cast<float>(x+1)/w);

        // (x*2,3) is formerly (x*2,2)
        U32 mTopNew = m->splitVert(m->eEdge(x*2,3), m->eVertPrev(m->eEdge(x*2,3)));
        m->verts[m->vertIdx(mTopNew)].pos = glm::mix(topL, topR, static_cast<float>(x+1)/w);
        m->verts[m->vertIdx(m->eVertPrev(mTopNew))].pos = glm::mix(topL, topR, static_cast<float>(x+1)/w);

        U32 mBotFlip = m->eVertNext(mBotNew);
        U32 mTopFlip = m->eVertNext(mTopNew);
        LOG("splitting with %d and %d, orig %d", mBotNew, mTopNew, m->eEdge(x*2,0));
        m->splitFace(mBotNew, mTopNew);
        m->splitFace(mTopFlip, mBotFlip);
        LOG("split face");
    }
    m->debugOut();
    LOG("");

    /* then slice horizontally from top to bottom */
    int nFaces = m->faces.size();
    for (int y = 1; y < h; ++y) {
        const int fBegin = (y-1)*nFaces, fEnd = fBegin + nFaces;
        const glm::vec3 rowL = glm::mix(m->verts[m->vertIdx(fBegin,0)].pos,
                                        m->verts[m->vertIdx(fBegin,3)].pos,
                                        static_cast<float>(y)/h),
                        rowR = glm::mix(m->verts[m->vertIdx(fEnd-1,1)].pos,
                                        m->verts[m->vertIdx(fEnd-1,2)].pos,
                                        static_cast<float>(y)/h);
        U32 mRowLeft = m->splitVert(m->eEdge(fBegin, 3), m->eVertPrev(m->eEdge(fBegin, 3)));
        m->verts[m->vertIdx(mRowLeft)].pos = rowL;
        m->verts[m->vertIdx(m->eVertPrev(mRowLeft))].pos = rowL;
        for (int x = 0; x < w; ++x) {
            LOG("%d", x);
            m->debugOut();
            const int fOld = (x*2)+fBegin;
            /* make a split to the left */
            const glm::vec3 thisR = glm::mix(rowL, rowR, static_cast<float>(x+1)/h);
            U32 mRightNew = m->splitVert(m->eEdge(fOld, 2), m->eVertPrev(m->eEdge(fOld, 2)));
            m->verts[m->vertIdx(mRightNew)].pos = thisR;
            m->verts[m->vertIdx(m->eVertPrev(mRightNew))].pos = thisR;
            m->splitFace(mRightNew, m->eEdge(fOld, 5));
            m->splitFace(m->eOpposite(m->eEdge(fOld, 5)), m->eOpposite(mRightNew));
        }
    }

    return m;
}
```

```cpp
#if 0
    for (int y = 0; y < h; ++y) {
        for (int x = 0; x < w; ++x) {
            m->verts.push_back();
        }
    }

    m->faces.resize(2);
    m->faces[0].count = sides;
    m->faces[1].count = sides;
    m->faces[0].material = 0;
    m->faces[1].material = 0;

    float curRot = 0.f, incRot = 2*PI / sides;
    for (unsigned slot = 0; slot < sides; slot++, curRot += incRot) {
        MeshVert &thisVert = m->verts[slot];
        thisVert.pos = glm::vec3(cosf(phase+curRot)*radius,
                                 sinf(phase+curRot)*radius,
                                 0.f);
        thisVert.texCoord = (glm::vec2(thisVert.pos) + glm::vec2(1.f,1.f)) / 2.f;

        if (i==max) {
            thisVert.normal = glm::vec3(0.f, 0.f, 1.f);
            MeshVert &otherVert = m->verts[sides + (sides-slot)/sides];
            otherVert.pos = thisVert.pos;
            otherVert.texCoord = thisVert.texCoord;

            m->faces[0].verts[slot] = slot;
            m->faces[1].verts[slot] = sides + slot;
        } else {
            thisVert.normal = thisVert.pos;
            m->faces[0].verts[slot] = slot;
            m->faces[1].verts[slot] = (sides-slot)/sides;
            m->faces[0].opposite[slot] = m->eEdge(1, sides-slot-1);
            m->faces[1].opposite[sides-slot-1] = m->eEdge(0, slot);
        }
    }
    ASSERT(m->check());
```

```cpp
void Mesh::extrudeFace(U32 faceIdx, float length)
{
    unsigned oldSize = faces.size();
    ASSERT(faceIdx < oldSize, "Mesh::extrude invalid faceIdx");

    MeshFace &face = faces[faceIdx];
    faces.resize(oldSize + face.count);
    for (U32 slot = 0; slot < face.count; slot++) {
        vert(face.verts[slot]).pos += length * face.normal;
        if (slot+1 == face.count) continue;
        MeshFace &newFace = faces[oldSize + slot];
        newFace.count = 4;
        newFace.material = face.material;
        // XXX: is this the right way of calculating the new face normals?
        newFace.normal = glm::vec3::cross(face.normal,
                                          verts[face.verts[slot+1]].pos -
                                          verts[face.verts[slot]].pos);
    }
}

void Mesh::extrude(U32 startFaceIdx, float length)
{
    unsigned oldSize = faces.size();
    ASSERT(startFaceIdx < oldSize, "Mesh::extrude invalid faceIdx");

    std::vector<U32> edgeList;        // list of boundary edges
    std::vector<U32> faceStack;       // stack for dfs
    faceStack.push_back(startFaceIdx);
    while (!faceStack.empty()) {
        U32 faceIdx = faceStack.back();
        faceStack.pop_back();
        for (U32 slot = 0; slot < faces[faceIdx].count; slot++) {
            U32 nextFaceIdx = faces[faceIdx].opposite[slot] / 8;
            if (faces[nextFaceIdx].selected == 1) {
                // visit this face only if selected and not visited before
                faces[nextFaceIdx].selected = 2; // selected but visited before
                faceStack.push_back(nextFaceIdx);
            }
            if (faces[nextFaceIdx].selected == 0) {
                // this is a boundary edge only if other face not selected
                edgeList.push_back(faceIdx*8 + slot);
            }
        }
    }
    for (unsigned edgeListIdx = 0; edgeListIdx < edgeList.size(); edgeListIdx++) {
        // make a face
        // set the parameters of the new face
    }
}
```

```cpp
U32 Mesh::dupVert(U32 baseEdge)
{
    ASSERT(faces[baseEdge/8].count < MAXVERT,
        "Mesh::dupVert on face which will have too many vertices");

    MeshFace &face = faces[baseEdge/8];
    for (unsigned slot = faces[baseEdge/8].count++; slot-- > baseEdge%8; ) {
        const unsigned newSlot = slot+1;
        face.verts[newSlot] = face.verts[slot];
        face.opposite[newSlot] = face.opposite[slot];
        // fix up opposite on adjacent face
        eOpposite(face.opposite[newSlot]) = eEdge(baseEdge/8, newSlot);
    }
    return eNext(baseEdge);
}

/* TODO: weight specifies how much the two split vertices are influenced by
 * the neighbouring vertices */
U32 Mesh::splitVert(U32 beginEdge, U32 endEdge/*, float weight*/)
{
    ASSERT(check());

    ASSERT(samePosVert(vertIdx(beginEdge), vertIdx(endEdge)),
        "Mesh::splitVert on two different vertices");
    U32 oldVertIdx = vertIdx(beginEdge);

    U32 newVertIdxBase = verts.size();

    // add new vertex to two affected faces
    dupVert(beginEdge);
    eOpposite(beginEdge) = endEdge;
    dupVert(endEdge);
    // there is a special case when the edges are the same
    eOpposite(endEdge) = beginEdge == endEdge ? 1 : 0);

    ASSERT(check());

    // update the vertex on affected faces
    U32 curEdge = eNext(beginEdge);
    U32 finEdge = eOpposite(beginEdge);
    std::map<int, int> vertMap;
    while (true) {
        ASSERT(vertIdx(curEdge) == oldVertIdx,
            "Mesh::splitVert on inconsistent mesh (not all half-edges "
            "originating from same vertex) for (face %u, slot %u) has %u but need %u",
            curEdge/8, curEdge%8, vertIdx(curEdge), oldVertIdx);
        auto vertMapIns = vertMap.insert(std::make_pair(vertIdx(curEdge),
                                         newVertIdxBase + vertMap.size()));
        vertIdx(curEdge) = vertMapIns.first->second;
        if (curEdge == finEdge) break;
        curEdge = eVertPrev(curEdge);
    }

    // create new vertices (as copies of old vertex)
    verts.insert(verts.end(), vertMap.size(), verts[oldVertIdx]);
    ASSERT(check());

    // return what used to be beginEdge, but has the new vertex
    return eNext(beginEdge);
}

#if 0
U32 Mesh::splitFace(U32 beginEdge, U32 endEdge)
{
    ASSERT(check());

    ASSERT(beginEdge/8 == endEdge/8, "meshSplitFace on two different faces");
    U32 oldFaceIdx = beginEdge/8;

    // create new face (as a copy of old face)
    U32 newFaceIdx = faces.size();
    faces.push_back(faces[oldFaceIdx]);

    MeshFace &oldFace = faces[oldFaceIdx];

    U32 begin = beginEdge%8, end = endEdge%8;
```

```cpp
    unsigned contFaceIdx = newFaceIdx,
             splitFaceIdx = oldFaceIdx;
    if (begin > end) {
        std::swap(begin, end);
        std::swap(contFaceIdx, splitFaceIdx);
    }
    /* contFace contains the continuous slots,
     * splitFace contains the slots that include the discontinuity */
    MeshFace &contFace = faces[contFaceIdx],
             &splitFace = faces[splitFaceIdx];

    unsigned faceCount = oldFace.count,
             contFaceCount = end - begin + 1,
             splitFaceCount = begin + faceCount - end + 1;

    contFace.count = contFaceCount;
    splitFace.count = splitFaceCount;

    for (unsigned slot = 0; slot < faceCount; slot++) {
        /* as oldFace.count == slot, it is always safe to read
         * oldFace.verts[slot] */
        if (begin <= slot && slot <= end) {
            unsigned thisSlot = contFace.count++;
            contFace.verts[thisSlot] = oldFace.verts[slot];
            U32 oppSlot = oldFace.opposite[slot];
            if (thisSlot != contFaceCount-1) {
                eOpposite(contFace.opposite[thisSlot]) = eEdge(contFaceIdx, thisSlot);
            }
        }
        if (slot <= begin || end <= slot) {
            unsigned thisSlot = splitFace.count++;
            splitFace.verts[thisSlot] = oldFace.verts[slot];
            splitFace.opposite[thisSlot] = oldFace.opposite[slot];
            if (thisSlot != begin) {
                eOpposite(splitFace.opposite[thisSlot]) = eEdge(splitFaceIdx, thisSlot);
            }
        }
    }
    ASSERT(contFace.count == contFaceCount);
    ASSERT(splitFace.count == splitFaceCount);

    eOpposite(eEdge(contFaceIdx, contFace.count-1)) = eEdge(splitFaceIdx, begin);
    eOpposite(eEdge(splitFaceIdx, begin)) = eEdge(contFaceIdx, contFace.count-1);
    ASSERT(check());

    return eEdge(newFaceIdx,
                 newFaceIdx == contFaceIdx ? contFace.count-1 : begin);
}
#endif
```

```cpp
U32 Mesh::splitFace(U32 beginEdge, U32 endEdge)
{
    ASSERT(check());

    ASSERT(beginEdge/8 == endEdge/8, "meshSplitFace on two different faces");
    ASSERT(beginEdge != endEdge, "meshSplitFace on same slots");

    const U32 oldFaceIdx = beginEdge/8,
              newFaceIdx = faces.size();
    // create new face (as a copy of old face)
    faces.push_back(faces[oldFaceIdx]);

    MeshFace &oldFace = faces[oldFaceIdx],
             &newFace = faces.back();
    const unsigned origFaceCount = oldFace.count;

    // set new face count
    newFace.count = (origFaceCount + end - begin) % origFaceCount + 1;
    oldFace.count = origFaceCount - 2 - newFace.count;

    // traverse from begin to end and move these into newFace
    for (U32 dstSlot = 0, srcSlot = (srcSlot+1 % origFaceCount) {
        ++dstSlot, srcSlot = (srcSlot+1) % origFaceCount) {
        newFace.opposite[dstSlot] = oldFace.verts[srcSlot];
        newFace.opposite[dstSlot] = oldFace.opposite[srcSlot];
        if (srcSlot == end) {
            ASSERT(dstSlot == newFace.count-1u);
        } else {
            eOpposite(newFace.opposite[dstSlot]) = eEdge(newFaceIdx, dstSlot);
        }
    }
    // fill in the gap in oldFace (if there is one)
    if (end > 0) {
        U32 dstStart = (end > begin ? begin+1 : 0);
        U32 srcEnd = (end > begin ? origFaceCount-1u : begin);
        for (U32 dstSlot = dstStart, srcSlot = end; ;
            ++dstSlot, ++srcSlot) {
            oldFace.verts[dstSlot] = oldFace.verts[srcSlot];
            oldFace.opposite[dstSlot] = oldFace.opposite[srcSlot];
            if (srcSlot != begin) {
                eOpposite(oldFace.opposite[dstSlot]) = eEdge(oldFaceIdx, dstSlot);
            }
            if (srcSlot == srcEnd) {
                ASSERT(dstSlot == oldFace.count-1u);
                break;
            }
        }
    }

    // fix up opposites on newly created edge
    U32 retEdge = beginEdge - (end > begin ? 0 : end);
    eOpposite(eEdge(newFaceIdx, newFace.count-1)) = retEdge;
    eOpposite(retEdge) = eEdge(newFaceIdx, newFace.count-1);

    return eOpposite(retEdge);
}
```

```
===
File: demolib/util.cpp (698 bytes)
===

#include "demolib.h"
#include <cstdio>
#include <cstdlib>
#include <cstdarg>
#include <ctime>
#include <execinfo.h>

void dlib_log(const char *file, unsigned line, const char *format, ...)
{
    va_list args;
    va_start(args, format);

    time_t secs = time(0);
    tm *t = localtime(&secs);
    fprintf(stderr, "[%04d-%02d-%02d %02d:%02d:%02d] %s:%d: ",
            t->tm_year+1900, t->tm_mon+1, t->tm_mday,
            t->tm_hour, t->tm_min, t->tm_sec,
            file, line);
    vfprintf(stderr, format, args);
    fprintf(stderr, "\n");

    va_end(args);
}

void dlib_stacktrace()
{
    void *array[16];
    size_t size = backtrace(array, 16);
    backtrace_symbols_fd(array, size, 2);
}
```

```
===
File: demolib/wave.cpp (2334 bytes)
===

#include "program.h"
#include "wave.h"

#include <cstring>
#include <cmath>

// major hack
const float Wave::WALL_VALUE = 1337.f;

Wave::Wave(unsigned _w, unsigned _h) :
    //Data(new Data[_w*_h]),
    data((Data *) dlib_malloc(_w*_h*sizeof(Data), 64)),
    w(_w), h(_h),
    moveDown(true)
{
    reset();
}

Wave::~Wave()
{
    //delete[] data;
    free(data);
}

void Wave::reset()
{
    /*
    for (unsigned y = 0; y < h; ++y) {
        for (unsigned x = 0; x < w; ++x) {
            D(x,y) = 0.f;
            Di(x,y) = 0.f;
        }
    }
    */
    memset(data, 0, sizeof(Data)*w*h);
}

void Wave::update()
{
    static const float theta = 0.3f;
    static const float sintheta = sinf(theta);
    static const float costheta = cosf(theta);
    static const unsigned edgedamp = 500;
    static const unsigned edgedtheta = 30;

    unsigned y = 1, yend = h-2, yinc = 1;
    if (!moveDown) {
        std::swap(y, yend); yinc = -1;
    }
    bool moveRight = moveDown;
    for (; y != yend; y += yinc) {
        unsigned x = 1, xend = w-2, xinc = 1;
        if (!moveRight) {
            std::swap(x, xend); xinc = -1;
        }
        for (; x != xend; x += xinc) {
            if (D(x, y)) {
                ASSERTX(D(x, y) == 0.f);
                continue;
            }
            float basis = (D(x, y-1) +
                           D(x-1, y) +
                           D(x+1, y) +
                           D(x, y+1)) / 4;

            unsigned dist = std::min(std::min(x, y), std::min(w-x-1, h-y-1));

            float re = D(x,y) - basis,
                  im = Di(x,y);
            if (dist < edgedamp) {
                re = re * (edgedamp-(edgedamp-dist)) / edgedamp;
                im = im * (edgedamp-(edgedamp-dist)) / edgedamp;
            }
            re *= 0.998f;
            im *= 0.998f;

            D(x,y) = basis + re*costheta - im*sintheta;
            Di(x,y) =         im*costheta + re*sintheta;
        }
        moveRight = !moveRight;
    }
    moveDown = !moveDown;
}

TextureWave::TextureWave(const Wave &_wave) :
    Texture2D(_wave.w, _wave.h, new uint8_t[_wave.w*_wave.h]),
    wave(_wave)
{
}

void TextureWave::update()
{
    unsigned end = h*w;
    for (unsigned i = 0; i < end; ++i) {
        data[i] = clamp(int(wave.data[i].v*127.f) + 128, 0, 255);
    }
    Texture2D::update();
}
```

```
===
File: demolib/pngwrite.cpp (3643 bytes)
===

#include <cstdio>
#include <png.h>
#include <functional>
#include "demolib.h"
#include "pngwrite.h"

static void error_handler(png_structp, png_const_charp msg)
{
    LOG("WARNING: libpng failed: %s", msg);
}

static bool DoPNGWrite(const char *fileName, uint8_t *data,
                       int w, int h, int stride,
                       std::function<void(png_structp, png_infop)> setMetadata)
{
    LOG("Writing PNG of %dx%d to '%s'", w, h, fileName);
    bool ret = false;
    png_structp png_ptr = NULL;
    png_infop info_ptr = NULL;

    FILE *fout = fopen(fileName, "wb");
    if (!fout) {
        LOG("WARNING: PNGWrite: could not open file '%s' for writing", fileName);
        goto cleanup;
    }

    png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, error_handler, NULL);
    if (!png_ptr) {
        LOG("WARNING: PNGWrite: could not create PNG struct");
        goto cleanup;
    }

    info_ptr = png_create_info_struct(png_ptr);
    if (!info_ptr) {
        LOG("WARNING: PNGWrite: could not create info struct");
        goto cleanup;
    }

    png_init_io(png_ptr, fout);
```

```cpp
    setMetadata(png_ptr, info_ptr);

    png_write_info(png_ptr, info_ptr);

    for (int row = 0; row < h; row++) {
        png_write_row(png_ptr, data + row*stride);
    }

    png_write_end(png_ptr, NULL);

    ret = true;

cleanup:
    if (png_ptr) {
        png_destroy_write_struct(&png_ptr, info_ptr ? &info_ptr : NULL);
    }
    fclose(fout);

    return ret;
}

bool PNGWrite(SDL_Surface *surf, const char *fileName)
{
    return DoPNGWrite(fileName, static_cast<uint8_t *>(surf->pixels), surf->w, surf->h, surf->pitch, [&](png_structp png_ptr,
png_infop info_ptr) {
        png_set_IHDR(png_ptr, info_ptr, surf->w, surf->h,
                     surf->format->BitsPerPixel,
                     pal ? PNG_COLOR_TYPE_PALETTE : PNG_COLOR_TYPE_RGB_ALPHA,
                     PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);

        if (pal) {
            png_color colors[256];
            int nColors = surf->format->palette->ncolors;
            for (int i = 0; i < nColors; ++i) {
                colors[i].red   = surf->format->palette->colors[i].r;
                colors[i].green = surf->format->palette->colors[i].g;
                colors[i].blue  = surf->format->palette->colors[i].b;
            }
            png_set_PLTE(png_ptr, info_ptr, colors, nColors);
        }
    });
}

template <typename T>
struct TextureData {
    static const int colorType;
    static const int bitDepth;
};

template <typename T> const int TextureData<T>::colorType = PNG_COLOR_TYPE_GRAY;
template <typename T> const int TextureData<T>::bitDepth = sizeof(T)*8;
template <> const int TextureData<glm::u8vec4>::colorType = PNG_COLOR_TYPE_RGB_ALPHA;
template <> const int TextureData<glm::u8vec4>::bitDepth = sizeof(int8_t)*8;

template <typename T>
bool PNGWrite(Texture2D<T> *tex, const char *fileName)
{
    int stride = tex->stride > 0 ? tex->stride : tex->w * sizeof(T);
    return DoPNGWrite(fileName, (uint8_t *)tex->data, tex->w, tex->h, stride, [&](png_structp png_ptr, png_infop info_ptr) {
        png_set_IHDR(png_ptr, info_ptr, tex->w, tex->h, stride,
                     TextureData<T>::bitDepth,
                     TextureData<T>::colorType,
                     PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    });
}

template bool PNGWrite(uint8_t[Texture2D<uint8_t> *surf, const char *fileName);
template bool PNGWrite(glm::u8vec4<Texture2D<glm::u8vec4> *surf, const char *fileName);
```

```
===
File: experiment/verletbridge.html (23859 bytes)
===

<!DOCTYPE html>
<!-- http://graphics.cs.cmu.edu/nsp/course/15-869/2006/papers/jakobsen.htm -->
<html>
<head>
<style type="text/css">
html, body {
    margin: 0;
}
canvas {
    background-color: white;
}
#info {
    font-weight: bold;
}
</style>
</head>
<body>
<div style="float: left; width: 580px; margin: 0; margin-right: 30px;">
    <canvas id="c" width="500" height="500">Your browser doesn't support canvas, you suck</canvas>
    <span style="float: left; font-size: 150%;">Current material: <span id="info">Loading...</span></span>
    <div style="clear: right;">
        <button onclick="setMode(new Simulator(builder))">Start</button>
        <button onclick="setMode(builder)">Stop</button>
    </div>
    <div style="clear: both; border-top: 1px solid black; margin-top: 10px; padding-top: 5px;">
        <textarea type="text" style="width: 100%;" id="saveload" autocomplete="off"></textarea>
        <button onclick="document.getElementById('saveload').value = builder.serialize()">Get build code</button>
        <button onclick="setMode(builder); builder.unserialize(document.getElementById('saveload').value);">Load from code</button>
        <button onclick="builder.reset();">Reset build</button>
    </div>
</div>
<h1>Cheng's unfinished Verlet integration demonstration</h1>
<h3>Instructions</h3>
<ul>
    <li>Press 0, 1, 2 (not on the numpad) to select between three materials</li>
    <li>Left click and drag from an existing point to build a bar of your material</li>
    <li>Right click when a bar is highlighted in red to delete it</li>
    <li>Right click and drag when a green point is highlighted to move it around</li>
    <li>Blue points are fixed points, which can't move. Use them as your starting point for your bridge</li>
</ul>
<h3>Challenges</h3>
<ul>
    <li>
        <tt>6 3 100 300 1 400 300 1 100 100 1 400 100 1 203 200 0 315 300 0 0 4 4 5 0 5 1 6</tt><br>
        Without moving the existing green points, can you save this iron platform from collapsing?
    </li>
    <li>Can you do the same without using cable?</li>
    <li>Can you do the same without using the top two fixed points?</li>
</ul>
<h3>Constructions</h3>
<ul>
    <li><tt>7 16 250 100 1 426 50 0 431 47 0 445 43 0 455 54 0 449 70 0 436 72 0 0 1 1 2 2 3 3 4 4 5 5 6 6 1 1 6 2 1 6 3 1</tt></li>
    <li>8 5 1 1 5 2 1 5 3 1 4 1 4 2 1 3 1</tt></li>
</ul>

<script type="text/javascript">
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}

function relMouseCoords(element, event) {
    var totalOffsetX = 0;
    var totalOffsetY = 0;
    var canvasX = 0;
    var canvasY = 0;
    do {
        totalOffsetX += element.offsetLeft - element.scrollLeft;
        totalOffsetY += element.offsetTop - element.scrollTop;
    } while (element = element.offsetParent);

    canvasX = event.pageX - totalOffsetX - document.body.scrollLeft;
    canvasY = event.pageY - totalOffsetY - document.body.scrollTop;

    return new Point(canvasX, canvasY);
}

var c = document.getElementById('c');
var ctx = c.getContext('2d');
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype = {
    clone: function () {
        return new Point(this.x, this.y);
    },
    add: function (p) {
        return new Point(this.x+p.x, this.y+p.y);
    },
    sub: function (p) {
        return new Point(this.x-p.x, this.y-p.y);
    },
    mul: function (n) {
        return new Point(this.x*n, this.y*n);
    },
    div: function (n) {
        return new Point(this.x/n, this.y/n);
    },
    dot: function (p) {
        return this.x*p.x + this.y*p.y;
    },
    lerp: function (p, i) {
        return new Point(this.x + (p-this.x)*i,
                         this.y + (p.y-this.y)*i);
    },
    cross: function (p) {
        return this.x*p.y - this.y*p.x;
    },
    ang: function (p) {
        return Math.atan2(this.cross(p), this.dot(p));
    },
    // the angle formed by the line a-this, and this-c
    ang2: function (a, c) {
        return a.sub(this).ang(c.sub(this));
    },
    rotAbout: function (p, theta) {
        var sin = Math.sin(theta),
            cos = Math.cos(theta),
            dx = this.x - p.x,
            dy = this.y - p.y;
        return new Point(cos*dx - sin*dy + p.x,
                         sin*dx + cos*dy + p.y);
    },
    len2: function () {
        return this.dot(this);
    },
    len: function () {
        return Math.sqrt(this.len2());
    },
    dist2: function (p) {
        return this.sub(p).len2();
    },
    dist: function (p) {
        return this.sub(p).len();
    },
    distLine: function (q1, q2) {
        var r = q2.sub(q1);
        var rlen2 = r.len2();
        if (rlen2 == 0) return q1.dist(this);
        var t = this.sub(q1).dot(r) / rlen2;
        if (t < 0) return q1.dist(this);
        if (t > 1) return q2.dist(this);
        return q1.lerp(q2, t).dist(this);
    },
};

function isIntersecting(p, p2, q, q2)
{
    var r = p2.sub(p), s = q2.sub(q);
    var diff = q.sub(p);
    var rxs = r.cross(s);
    if (Math.abs(rxs) < 1e-7) return false;
    var qSol = diff.cross(s) / rxs;
    var qSol = diff.cross(r) / rxs;
    return (pSol >= 0 && pSol <= 1 && qSol >= 0 && qSol <= 1);
}

var materials = [
    {
        name: "Brittle Iron",
        lengthFactor: 1,
        angFixed: 1.00,
        angLimit: 4 * Math.PI / 100,
        linearStiffness: 1,
        tensionLimit: 1.03,
        compressionLimit: 0.99,
        density: 2.3,
        colour: "#595050"
    },
    {
        name: "Steel",
        lengthFactor: 1,
        angFixed: 0.6,
        angLimit: 4 * Math.PI / 100,
        linearStiffness: 1,
        tensionLimit: 1.025,
        compressionLimit: 0.98,
        density: 2,
        colour: "#C0C0C0"
    },
    {
        name: "Cable",
        lengthFactor: 1,
        angFixed: 0,
        angLimit: 360 * Math.PI / 180,
        linearStiffness: 1,
        tensionLimit: 1.05,
        compressionLimit: 0.0,
        density: 1,
        colour: "#D03DA0"
    },
];

function Node(p) {
    this.p = p;
    this.prevP = p.clone();
    this.fixed = false;
    this.finalised = false;
    this.links = [];
}
Node.prototype = {
    clone: function () {
        node = new Node(this.p.clone());
        node.fixed = this.fixed;
        node.finalised = this.finalised;
        node.links = [];
        return node;
    }
};
function Link(n1, n2) {
    this.material = null;
    this.n1 = n1;
    this.n2 = n2;
    this.selected = false;
    if (n1) n1.links.push(this);
    if (n2) n2.links.push(this);
}

function NodeBase() {
    this.nodes = [];
```

```
        this.links = [];
        this.drawStyle = {
            background: "#FFFFFF",
            drawFinalisedNodes: true,
            drawMaterialStress: false
        };
    }
    NodeBase.prototype = {
        start: function () {this.draw();},
        stop: function () {},
        onMouseDown: function () {},
        onMouseMove: function (x, y) {},
        onMouseUp: function () {},
        onKeyPress: function (c) {},
        draw: function () {
            ctx.fillStyle = this.drawStyle.background;
            ctx.fillRect(0,0,c.width,c.height);
```

...

```
===
File: experiment/cube.html (6036 bytes)
===
```

...

```
===
File: experiment/verlet.html (2907 bytes)
===
```

...

```
===
File: experiment/verletrag.html (6135 bytes)
===
```

```javascript
            var dir = points[link[1]].sub(points[link[0]]);
            var len = dir.len();
            var fact = (len-link[2])/len;
            var correction = dir.mul(fact/2);
            points[link[0]] = points[link[0]].add(correction);
            points[link[1]] = points[link[1]].add(correction.mul(-1));
        }
    }
    // verlet
    for (var i = 0; i < points.length; ++i) {
        var newPoint = points[i].mul(2).sub(prevPoints[i]).add(new Point(0,1));
        prevPoints[i] = points[i];
        points[i] = newPoint;
    }
    for (var i = 0; i < bloods.length; ++i) {
        var newBlood = bloods[i].mul(2).sub(prevBloods[i]).add(new Point(0,1));
        prevBloods[i] = bloods[i];
        bloods[i] = newBlood;
        if (!(bloods[i].x >= 0 && bloods[i].x <= 500 && bloods[i].y >= 0 && bloods[i].y <= 500)) {
            bloods.splice(i, 1);
            prevBloods.splice(i, 1);
            --i;
        }
    }
}

function randomObstacle() {
    return new Point(Math.floor(Math.random() * 500), 0);
}

// add obstacles
if (tick % 33 == 0) {
    obstacles.push(randomObstacle());
}
for (var i = 0; i < obstacles.length; ++i) {
    var oldObstacle = obstacles[i].clone();
    obstacles[i].y += 5;
    if (obstacles[i].y > 500) {
        obstacles[i] = randomObstacle();
    } else {
        for (var k = 0; k <= 1; k += 0.1) {
            for (var j = 0; j < links.length; ++j) {
                if (isIntersecting(points[links[j][0]].lerp(prevPoints[links[j][0]], k),
                                   points[links[j][1]].lerp(prevPoints[links[j][1]], k),
                                   oldObstacle, obstacles[i])) {
                    for (var z = 0; z <= 1; z += 0.05) {
                        var x = [Math.random() * 2*Math.PI, Math.random() * 2*Math.PI];
                        var blood = points[links[j][0]].lerp(points[links[j][1]], z);
                        bloods.push(blood);
                        prevBloods.push(new Point(blood.x - 10*Math.cos(x[0]), blood.y - 3*Math.cos(x[1])));
                    }
                    links.splice(j, 1);
                    --j;
                }
            }
        }
    }
}

// draw limbs
ctx.strokeStyle = '#FFFFFF';
for (var i = 0; i < links.length; ++i) {
    ctx.beginPath();
    //ctx.strokeStyle = 'hsla(' + (i/links.length*360) + ', 100%, 50%, 0.2)';
    ctx.moveTo(points[links[i][0]].x, points[links[i][0]].y);
    ctx.lineTo(points[links[i][1]].x, points[links[i][1]].y);
    ctx.stroke();
}

// draw head
ctx.fillStyle = '#FFFFFF';
ctx.beginPath();
ctx.arc(points[0].x, points[0].y, 20, 0, 2*Math.PI, true);
ctx.fill();

// draw obstacles
ctx.fillStyle = '#000000';
for (var i = 0; i < obstacles.length; ++i) {
    ctx.beginPath();
    ctx.arc(obstacles[i].x, obstacles[i].y, 2, 0, 2*Math.PI, true);
    ctx.fill();
}

// draw blood
ctx.fillStyle = '#FF0000';
for (var i = 0; i < bloods.length; ++i) {
    ctx.beginPath();
    ctx.arc(bloods[i].x, bloods[i].y, 2, 0, 2*Math.PI, true);
    ctx.fill();
}

tick += 1;
}, 50);

</script>
<div>
<button onclick="location.reload()">Restart?</button>
</div>
</body>
</html>
```

```
===
File: experiment/points.html (3104 bytes)
===
```

```html
<!DOCTYPE html>
<html>
<head>
<style type="text/css">
canvas {
    background-color: white;
}
</style>
</head>
<body id="b">
<canvas id="c" width="500" height="500"></canvas>
<script type="text/javascript">
var c = document.getElementById('c');
var ctx = c.getContext('2d');
ctx.translate(c.width/2,c.height/2);
ctx.scale(c.width/4,c.height/4);
function Point(x, y) {
    this.x = x;
    this.y = y;
}
function matrixProduct(a, b) {
    var ca = b[0].length,
        cb = a.length,
        cc = a[0].length;
    if (ca != b.length) throw new Error("dimension mismatch");
    c = [];
    c.length = cb;
    for (var cr = 0; cr < cb; ++cr) {
        c[cr] = [];
        c[cr].length = ca;
        for (var cc = 0; cc < ca; ++cc) {
            c[cr][cc] = 0;
            for (var cj = 0; cj < cc; ++cj) {
                c[cr][cc] += a[cr][cj] * b[cj][cc];
            }
        }
    }
    return c;
}
// matrix product for the special case where p is a column vector expressed as an array
function transformProduct(m, p) {
    if (m.length != p.length) throw new Error("dimension mismatch");
    var outp = []; outp.length = p.length;
    for (var r = 0; r < m.length; ++r) {
        outp[r] = 0;
        for (var j = 0; j < p.length; ++j) {
            outp[r] += m[r][j] * p[j];
        }
    }
    return outp;
}
function scaleVector(v, s) {
    return [v[0]*s, v[1]*s, v[2]*s];
}
function vectorProduct(a, b) {
    return [a[1]*b[2]-a[2]*b[1], a[2]*b[0]-a[0]*b[2], a[0]*b[1]-a[1]*b[0]];
}
function scalarProduct(a, b) {
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
function modulus(a) {
    return Math.sqrt(a[0]*a[0] + a[1]*a[1] + a[2]*a[2]);
}

var tick = 0;

var points = [];
for (var i = 0; i < 1000; ++i) {
    var a = [Math.random() * 2*Math.PI, Math.random() * 2*Math.PI, Math.random() * 2*Math.PI];
    points.push([[Math.cos(a[0])], [Math.sin(a[1])], [Math.cos(a[2])]]);
}

setInterval(function () {
    ctx.fillStyle = 'rgba(255,255,255, 0.5)';
    ctx.fillRect(-2,-2,-4,4);

    var rotx = tick * 0.14, roty = tick * 0.17, s = Math.sin(tick/10)/2+.75;
    var mx =
        [[1,0,0],
         [0, Math.cos(rotx), -Math.sin(rotx)],
         [0, Math.sin(rotx), Math.cos(rotx)]];
    var my =
        [[Math.cos(roty),0,Math.sin(roty)],
         [0, 1, 0],
         [-Math.sin(roty),0 , Math.cos(roty)]];
    var m =
        [[s, 0, 0],
         [0, s, 0],
         [0, 0, s]];
    var rotm = matrixProduct(mx, my);
    var mm = matrixProduct(m, rotm);

    for (var i = 0; i < points.length; ++i) {
        var pointmod = modulus(points[i]);
        var scale = 1/Math.pow(pointmod, 1-s);
        var warpedpoint = scaleVector(points[i], scale);
        var finalpoint = transformProduct(mm, warpedpoint);
        ctx.fillStyle = 'hsla(' + (i/points.length*360) + ', 100%, 50%, ' + (0.3-finalpoint[2]/5) + ')';
        ctx.beginPath();
        ctx.arc(finalpoint[0], finalpoint[1], s*0.05, 0, 2*Math.PI, true);
        ctx.fill();
    }
    tick += 1;
}, 50);
</script>
</body>
</html>
```

```
===
File: include/program.h (7190 bytes)
===
```

```cpp
#ifndef DEMOLIB_PROGRAM_H
#define DEMOLIB_PROGRAM_H

#include "mesh.h"
#include <GL/gl.h>
#include <vector>

struct TextureBase
{
    TextureBase();
    ~TextureBase();

    bool allocated;         // if glTexImage*D already called (via allocate())
                            // to allocate memory on the GPU

    // bind texture without updating
    virtual void bind(GLuint texUnit) const = 0;
    // allocate and update texture (only called once), needs to be bound first
    virtual void allocate() = 0;
    // update texture, needs to be bound first
    virtual void update() = 0;
    // get texture from GPU
    virtual void updateLocal() = 0;

    static bool isInUseId(GLuint id) {return currentId() == id;}
    bool isInUse() const {return isInUseId(id);}
    static GLuint currentId();

    GLuint id;
};

struct Texture2DBase : public TextureBase
{
    Texture2DBase(int _w, int _h, int _stride) : w(_w), h(_h), stride(_stride) {}

    int w, h;
    int stride;     // can be set to 0

    void bind(GLuint texUnit) const;
    void unbind() const;
};

/* 2D texture with optionally automatically managed memory */
template <typename T>
struct Texture2D : public Texture2DBase
{
    /* pass all defaults to not initialise anything:
     *     autoManaged is set to false, nothing done on destruction
     * pass width/height but not data
     *     autoManaged is set to true, data is automatically allocated and
     *     destroyed
     * pass width height and data
     *     autoManaged is set to false
     */
    // constructor to do nothing: manually set everything
    Texture2D() :
        Texture2DBase(-1, -1, 0), data(nullptr),
        autoManaged(false)
    {}
    // constructor to auto-manage memory: manually set nothing
    Texture2D(int _w, int _h) :
        Texture2DBase(_w, _h, 0), data(new T[_w*_h]),
        autoManaged(true)
    {}
    // constructor to manually-manage memory from an external source:
    // memory will not be deallocated on destruction
    Texture2D(int _w, int _h, T *_data, int _stride = 0) :
        Texture2DBase(_w, _h, _stride), data(_data),
        autoManaged(false)
    {}

    virtual ~Texture2D()
    {
        if (autoManaged) {
            delete[] data;
        }
    }

    T *const data;
    const bool autoManaged; // if we need to deallocate memory on destruction

    virtual void allocate();
    virtual void update();
    virtual void updateLocal();

    static const GLenum pixelType, pixelFormat, internalPixelFormat;
};
```

```
===
File: include/demolib.h (124 bytes)
===
```

```cpp
#ifndef DEMOLIB_H
#define DEMOLIB_H

#include <GL/glew.h>
#include "util.h"
#include "mesh.h"
#include "program.h"

#endif
```

```
===
File: include/text.h (757 bytes)
===
```

```cpp
#ifndef DEMOLIB_TEXT_H
#define DEMOLIB_TEXT_H

#include "program.h"
#include <string>

struct SDL_Surface;
typedef struct _TTF_Font TTF_Font;

struct Font
{
    Font(std::string file, int size);
    ~Font();

    SDL_Surface* measure(std::string text);
    SDL_Surface *draw(std::string text);

private:
    TTF_Font *font;
};

struct TextureTextSDF : public Texture2Duint8_t>
{
    TextureTextSDF(SDL_Surface *surf, int scaleDown = 16, int spread = 32);
    TextureTextSDF(int w, int h, SDL_Surface *surf, int spread = 32);
};

struct ProgramTextSDF : public ProgramTexQuadImp
{
    ProgramTextSDF(GLuint texInit,
                   std::initializer_list<std::shared_ptr<Shader> > && shaders = {vs, fs});

    static const std::shared_ptr<Shader> fs;
};

#endif
```

```
===
File: include/util.h (2882 bytes)
===
```

```cpp
#ifndef DEMOLIB_UTIL_H
#define DEMOLIB_UTIL_H

#include <memory>
#include <cinttypes>
#include <vector>

typedef uint8_t U8;
typedef uint16_t U16;
typedef uint32_t U32;
static constexpr U32 U32_MIN = static_cast<U32>(0);
static constexpr U32 U32_MAX = ~static_cast<U32>(0);
typedef int8_t S8;
typedef int16_t S16;
typedef int32_t S32;

#include <glm/glm.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtc/type_precision.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

/* matrix stack */

template <class M>
struct MatStack
{
    void push() {
        stack.push_back(top);
    }
    void pop() {
        top = stack.back();
```
(continued)
```cpp
        stack.pop_back();
    }
    MatStack &operator*=(const M &m) {
        top *= m;
        return *this;
    }

    std::vector<M> stack;
    M top;
};

template struct MatStack<glm::mat4>;
typedef MatStack<glm::mat4> Mat4Stack;

#define ARRAYLEN(x) (sizeof(x)/sizeof((x)[0]))
#define PI 3.14159265359f

template <typename T>
inline T clamp(const T x, const T min, const T max)
{
    return std::max(min, std::min(x, max));
}

extern "C"
{
    void dlib_log(const char *file, unsigned line, const char *format, ...);
    void dlib_stacktrace();
}

#ifdef LOG
#undef LOG
#endif

#ifdef ASSERTX
#undef ASSERTX
#endif

#ifdef INFO
#undef INFO
#endif

#ifdef DEBUG
    static constexpr bool IS_DEBUG = true;
#   include <cstdlib>
#   define LOG(...) INFO(__VA_ARGS__)
#   define ASSERTX(x, ...) \
    do { \
        if (!(x)) { \
            LOG("** ASSERTION '%s' FAILED! **", #x); \
            LOG(__VA_ARGS__); \
            dlib_stacktrace(); \
            abort(); \
        } \
    } while (0)
#   define CHECK_GL_ERROR(where) \
    do { \
        GLenum glErr = glGetError(); \
        if (glErr) { \
            LOG("GL error %s", gluErrorString(glErr)); \
            ASSERTX(glErr, "GL error detected (%s", where); \
        } \
    } while (0)
#else
    static constexpr bool IS_DEBUG = false;
#   define LOG(...)     do { (void) 0; } while (0)
#   define ASSERTX(x, ...) do { (void) sizeof(x); } while (0)
#   define INFO(...)   do { (void) sizeof(where); } while (0)
#   define CHECK_GL_ERROR(where) do { (void)0; sizeof(where); } while (0)
#endif

#include <cerrno>
#include <cstdlib>
inline void *dlib_malloc(size_t size, size_t alignment)
{
    void *ptr;
    int ret = posix_memalign(&ptr, alignment, size);
    ASSERTX(ret != ENOMEM, "alignment invalid");
    ASSERTX(ret = ENOMEM, "insufficient memory");
    ASSERTX(ret == 0, "posix_memalign failed -- unknown reason");
    return ptr;
}

#include <xmmintrin.h>
#include <xmmintrin.h>

#define ENABLE_FTZ() _mm_setcsr(_mm_getcsr() | 0x8040)

#define likely(expr)   __builtin_expect(!!(expr), 0)
#define unlikely(expr) __builtin_expect(!!(expr), 1)

template <class T>
T lerp(T a, T b, int num, int den)
{
    return a + (b-a)*num/den;
}

#endif
```

```
===
File: include/scene.h (326 bytes)
===
```

```cpp
#ifndef DEMOLIB_SCENE_H
#define DEMOLIB_SCENE_H

#include "util.h"
#include "mesh.h"
#include "program.h"

struct SceneNode
{
    Mesh *mesh;
    Program *program;
    glm::vec3 *pos;
    glm::quat *orientation;
    float scale;

    TextureBase *texture;
};

struct Scene
{
    Scene();

    void Render() const;
};

#endif
```

```
===
File: include/demo.h (278 bytes)
===
```

```cpp
#ifndef DEMO_H
#define DEMO_H

#include "demolib.h"

extern "C" {
    void demo_init(unsigned w, unsigned h);
    bool demo_prepareFrame();
    void demo_drawFrame();

    void demo_evtMouseMove(int x, int y);
    void demo_evtMouseButton(uint8_t button, bool state);
}

#endif
```

```
===
File: include/mesh.h (5168 bytes)
===
```

```cpp
#ifndef MESH_H
#define MESH_H

#include "util.h"
#include <vector>

static constexpr unsigned MAXVERT = 8;   // max vertices per face

struct MeshVert
{
    U8 mask;
    U32 tmp;
    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec2 texCoord;
};

struct MeshFace
{
    U8 count;            // number of verts/edges
    U8 material;         // material index, 0 for none
    U8 mask;
    U32 tmp;

    U32 verts[MAXVERT];      // vertex indices
    U32 opposite[MAXVERT];   // half-edge which is opposite
};

struct Mesh
{
    std::vector<MeshVert> verts;
    std::vector<MeshFace> faces;

    Mesh() {
    }

    Mesh(Mesh &&other) {
        verts = other.verts;
        faces = other.faces;
    }

    // basic operations
    static std::unique_ptr<Mesh> createRing(unsigned sides, bool crease, float phase = 0.f, float radius = 1.f);
    static std::unique_ptr<Mesh> createGrid(int w, int h);
    static std::unique_ptr<Mesh> createGrid(int w, int h);

    bool samePosVert(U32 v1, U32 v2) const {
        return (v1 == v2 || verts[v1].pos == verts[v2].pos);
    }

    bool check() const;
    /* meshCheckFlags checks a mesh to ensure that the selected flags are all
     * reset
     */
    bool checkFlags() const;

    void debugDot() const;

    void extrudeFace(U32 faceIdx, float length);

    /* dupVert (internally used) duplicates the slot before baseEdge
     *   v0 --->-- v1
     * dupVertex(e) creates a new edge e0 and a duplicate slot.
     *   v0 --e0-->- v0 --->-- v1
     * the new value of e is returned. e0 is equal to baseEdge after the operation.
     */
    U32 dupVert(U32 baseEdge);

    /* splitVert performs a vertex split
     * beginEdge - specifies the begin half-edge (inclusive)
     * endEdge - specifies the end half-edge (exclusive)
     * note that the order of edge traversal is CLOCKWISE (i.e. eVertPrev)!
     * returns the half-edge representing the new vertex (on the old face)
     */
    U32 splitVert(U32 beginEdge, U32 endEdge);

    /* splitFace performs a face split
     * beginEdge - specifies the begin half-edge
     * endEdge - specifies the end half-edge
     * going anticlockwise (eNext) from beginEdge to endEdge, those edges are
     * moved into the new face
     * returns the half-edge representing the new face which has all vertices
     * between beginEdge and endEdge
     */
    U32 splitFace(U32 beginEdge, U32 endEdge);

    // traversal functions
    inline U32 edge(U32 face, U32 slot) const {
        ASSERTX(face < faces.size());
        ASSERTX(slot < faces[face].count);

        return face*8 + slot;
    }

    inline bool eCheck(U32 face, U32 slot) const {
        if (face >= faces.size()) {
            LOG("eCheck failed for (face %u, slot %u): invalid face index (%u faces)",
                face, slot, faces.size());
            return false;
        }
        if (slot >= faces[face].count) {
            LOG("eCheck failed for (face %u, slot %u): invalid slot index (%u slots)",
                face, slot, faces[face].count);
            return false;
        }
        return true;
    }
    inline bool eCheck(U32 edge) const {
        return eCheck(edge/8, edge%8);
    }

    inline U32 ePrev(U32 face, U32 slot) const {
        ASSERTX(eCheck(face, slot));
        const U8 slotCount = faces[face].count;
        return (face*8) + ((slot+slotCount-1) % slotCount);
    }
    inline U32 ePrev(U32 edge) const {
        return ePrev(edge/8, edge%8);
    }

    inline U32 eNext(U32 face, U32 slot) const {
        ASSERTX(eCheck(face, slot));
        const U8 slotCount = faces[face].count;
        return (face*8) + ((slot+1) % slotCount);
    }
    inline U32 eNext(U32 edge) const {
        return eNext(edge/8, edge%8);
    }

    inline U32 eVertPrev(U32 face, U32 slot) const {
        ASSERTX(eCheck(face, slot));
        return eOpposite(ePrev(face, slot));
    }
    inline U32 eVertPrev(U32 edge) const {
        ASSERTX(eCheck(edge));
        return eVertPrev(edge/8, edge%8);
    }

    inline U32 eVertNext(U32 face, U32 slot) const {
        ASSERTX(eCheck(face, slot));
```

```cpp
struct Shader
{
public:
    Shader(GLenum _type, const char *_name);
    ~Shader();

    bool compile();
    bool compileLazy() {
        if (id == -1u) return compile();
        return true;
    }

    GLint id;
    GLenum type;
    const char *name, *strType;
    std::string code;

    static std::shared_ptr<Shader> FromFile(GLenum type, const char *filename);
    static std::shared_ptr<Shader> Inline(GLenum type, std::string shaderCode, const char *name = "<inline>");
};

struct Program
{
    static constexpr U32 PrimitiveRestartIndex = 0xFFFFFFFF;

protected:
    Program(std::initializer_list<std::shared_ptr<Shader> > && shaders);

public:
    virtual void draw() const = 0;
    bool link();
    virtual void use();
    virtual void unuse();
    /* checks */
    static GLuint currentId();
    static bool isInUseId(GLuint id) {return currentId() == id;}
    bool isInUse() const {return isInUseId(id);}

    GLint uniformLocation(const char *name);
    bool setUniform(const char *name, GLint v);
    bool setUniform(const char *name, float v);
    bool setUniform(const char *name, const glm::vec2 &v);
    bool setUniform(const char *name, const glm::vec3 &v);
    bool setUniform(const char *name, const glm::vec4 &v);
    bool setUniform(const char *name, const glm::mat2 &v, bool trans=false);
    bool setUniform(const char *name, const glm::mat3 &v, bool trans=false);
    bool setUniform(const char *name, const glm::mat4 &v, bool trans=false);

    GLuint id;

protected:
    /* called once link succeeds -- set up uniforms here */
    virtual void postLink() {}

private:
    std::vector<std::shared_ptr<Shader> > shaders;
};

struct VertexAttribDescriptor
{
    const char *shaderAttrName;
    unsigned offset;
};

struct ProgramRaw : public Program
{
    ProgramRaw(std::initializer_list<std::shared_ptr<Shader> > && shaders) :
        Program(std::move(_shaders)) {}

private:
    virtual void draw() const {};
};

struct VertBuf
{
    glm::vec3 pos;
    glm::vec2 texCoord;
    VertBuf &operator=(const MeshVert &vert) {
        pos = vert.pos;
        texCoord = vert.texCoord;
        return (*this);
    }
};

struct ProgramMesh : public Program
{
    ProgramMesh(std::initializer_list<std::shared_ptr<Shader> > && shaders) :
        Program(std::move(_shaders)) {}

    virtual void use();
    virtual void unuse();

    /* accepts vertices iff (vert.mask | vertMask) == 0xFF */
    virtual void updateMeshBuf(const Mesh &mesh, uint8_t vertMask = 0xFF, uint8_t faceMask = 0xFF);
    /* draw after doing sanity checks that this program is being used */
    virtual void draw() const;
    virtual void drawWire() const;
    /* example:
     *   prog->link();
     *   ...
     *   prog->use();
     *   prog->draw();
     *   prog->unuse();
     */

    GLuint vaoId, vertBufId, idxBufId;
    std::vector<VertBuf> vertBuf;
    std::vector<U32> idxBuf;

    static const std::shared_ptr<Shader> vs;

private:
    void doDraw(GLenum mode) const;
};

struct ProgramMeshDebugVisFace : public ProgramMesh
{
    ProgramMeshDebugVisFace() :
        ProgramMesh{vs, fs}
    {}
    static const std::shared_ptr<Shader> fs;
};

struct ProgramMeshDebugVisEdge : public ProgramMesh
{
    ProgramMeshDebugVisEdge() :
        ProgramMesh{vs, fs}
    {}
    static const std::shared_ptr<Shader> fs;
};

struct ProgramTexturedQuad : public ProgramMesh
{
    ProgramTexturedQuad(GLuint texInit,
        std::initializer_list<std::shared_ptr<Shader> > && shaders = {vs, fs});

    GLuint texInit;
    static const std::shared_ptr<Shader> vs, fs;

protected:
    virtual void postLink();
};

struct ProgramTexQuadImp : public Program
{
    ProgramTexQuadImp(GLuint texInit,
        std::initializer_list<std::shared_ptr<Shader> > && shaders = {vs, fs});

    virtual void draw() const;

    GLuint texInit, vaoId;
    static const std::shared_ptr<Shader> vs, fs;

protected:
    virtual void postLink();
};

struct FBOBase
{
};

struct FBODefault : public FBOBase
{
};

struct FBO : public FBOBase
{
    const int w, h;
    GLuint id, depthBufId;

    enum ConstructionFlags {
        DEPTHBUF = 1,
    };
    FBO(int _w, int _h, int flags = 0);
    virtual ~FBO();

    // bind a texture to draw to
    void bindTexture(const Texture2DBase &tex, int attachment);
};

struct GLContext
{
    static GLContext &get()
    {
        static GLContext instance;
        return instance;
    }

    void bindFBO(FBO &fbo);
    void unbindFBO();

    FBODefault defaultFBO;
    FBOBase *curFBO;
private:
    GLContext() : curFBO(0) {}
    GLContext(GLContext const&);
    void operator=(GLContext const&);
};

#endif
```

```cpp
            return eOpposite(ePrev(face, slot));
    }
    inline U32 eVertNext(U32 edge) const {
        ASSERT(eCheck(edge));
        return eOpposite(ePrev(edge));
    }

    inline U32 GeOpposite(U32 face, U32 slot) {
        ASSERT(eCheck(face, slot));
        return faces[face].opposite[slot];
    }
    inline U32 GeOpposite(U32 edge, U8 edge%8) {
        return eOpposite(edge/8, edge%8);
    }

    inline U32 eOpposite(U32 face, U32 slot) const {
        ASSERT(eCheck(face, slot));
        return faces[face].opposite[slot];
    }
    inline U32 eOpposite(U32 edge) const {
        return eOpposite(edge/8, edge%8);
    }

    inline U32 GvertIdx(U32 face, U32 slot) {
        ASSERT(eCheck(face, slot));
        return faces[face].verts[slot];
    }
    inline U32 GvertIdx(U32 edge) {
        return vertIdx(edge/8, edge%8);
    }

    inline U32 vertIdx(U32 face, U32 slot) const {
        ASSERT(eCheck(face, slot));
        return faces[face].verts[slot];
    }
    inline U32 vertIdx(U32 edge) const {
        return vertIdx(edge/8, edge%8);
    }
};

#endif
```

```
===
File: include/pngwrite.h (236 bytes)
===
```

```cpp
#ifndef DEMOLIB_PNGWRITE_H
#define DEMOLIB_PNGWRITE_H

#include <SDL.h>
#include "program.h"

bool PNGWrite(SDL_Surface *surf, const char *fileName);
template <typename T>
bool PNGWrite(Texture2D<T> *tex, const char *fileName);

#endif
```

```
===
File: include/wave.h (984 bytes)
===
```

```cpp
#ifndef DEMOLIB_WAVE_H
#define DEMOLIB_WAVE_H

#include "util.h"
#include "program.h"
#include <memory>
#include <limits>

struct Wave
{
    Wave(unsigned w, unsigned h);
    ~Wave();

    void reset();
    void update();

    float &D(unsigned x, unsigned y) {
        ASSERT(x < w && y < h);
        return data[x + y*w].v;
    }
    float &Di(unsigned x, unsigned y) {
        ASSERT(x < w && y < h);
        return data[x + y*w].i;
    }
    static const float WALL_VALUE;
    bool W(unsigned x, unsigned y) {
        ASSERT(x < w && y < h);
        return Di(x, y) == WALL_VALUE;
    }
    void Wset(unsigned x, unsigned y, bool v) {
        ASSERT(x < w && y < h);
        Di(x, y) = v ? WALL_VALUE : 0.0f;
    }

    struct Data {
        float v, i;
    };

    // row-major
    Data *data;

    unsigned w, h;
    bool moveDown;
};

struct TextureWave : public Texture2D<uint8_t>
{
    TextureWave(const Wave &wave);
    void update();

    const Wave &wave;
};

#endif
```

```
===
File: player/player.cpp (6687 bytes)
===
```

```cpp
#include "demolib.h"

#include <GL/glew.h>
#include <SDL.h>
#include <SDL_ttf.h>
#include <cstdio>
#include <ctime>
#include <getopt.h>

#define defsym(required, symname, ret, ...) \
    static ret (*symname)(__VA_ARGS__) = 0;
#include "defsym.h"
#undef defsym

#if defined(WINDOWS)

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <memory>

class LocalFree_deleter {
public:
    template <class T>
    void operator()(T *p) {
        LocalFree(p);
    }
};

static std::unique_ptr<char, LocalFree_deleter> formattedError(DWORD err)
{
    char *lpMsgBuf;
    FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        err,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        reinterpret_cast<LPSTR>(&lpMsgBuf),
        0, NULL);
    return std::unique_ptr<char, LocalFree_deleter>(lpMsgBuf);
}

#define defsym(required, symname, ret, ...) \
    symname = reinterpret_cast<ret (*)(__VA_ARGS__)>(GetProcAddress(demoLib, #symname)); \
    if (required && !symname) { \
        LOG("ERROR: Couldn't load demo library \"%s\": %s", libName, #symname, libName); \
        missingSymbol = true; \
    }
static void pl_loadDemoLib(const char *libName)
{
    HMODULE demoLib = LoadLibrary(libName);
    if (!demoLib) {
        LOG("ERROR: Couldn't load demo library \"%s\": %s", libName, formattedError(GetLastError()).get());
        exit(1);
    }

    bool missingSymbol = false;
#   include "defsym.h"

    if (missingSymbol) exit(1);
}
#undef defsym

#elif defined(LINUX)

#include <dlfcn.h>

#define defsym(required, symname, ret, ...) \
    symname = reinterpret_cast<ret (*)(__VA_ARGS__)>(dlsym(demoLib, #symname)); \
    if (required && !symname) { \
        LOG("ERROR: Symbol \"%s\" undefined in demo library \"%s\", %s", #symname, libName); \
        missingSymbol = true; \
    }
static void pl_loadDemoLib(const char *libName)
{
    void *demoLib = dlopen(libName, RTLD_NOW);
    if (!demoLib) {
        LOG("ERROR: Couldn't load demo library \"%s\": %s", libName, dlerror());
        exit(1);
    }

    bool missingSymbol = false;
#   include "defsym.h"

    if (missingSymbol) exit(1);
}
#undef defsym

#else
#   error "Compiling for unknown platform"
#endif

static SDL_Surface *screen;

static void pl_setVideoMode(int w, int h)
{
    LOG("Setting %dx%d GL video mode", w, h);
    screen = SDL_SetVideoMode(w, h, 32, SDL_OPENGL);
    if (screen)
        LOG("Set %dx%d GL video mode", screen->w, screen->h);
    } else {
        LOG("ERROR: Couldn't set %dx%d GL video mode: %s", w, h, SDL_GetError());
        exit(1);
    }
}

static const char *demoLibrary = "demo.so";

int main(int argc, char *argv[])
{
    LOG("Player starting");

    LOG("Parsing switches");
    int c, opt_idx;
    static const struct option long_opts[] = {
        {"help", no_argument, 0, 'h'},
        {"demo-library", required_argument, 0, 'd'}
    };
    while ((c = getopt_long(argc, argv, "hd:", long_opts, &opt_idx)) != -1) {
        switch (c) {
        case 'd':
            demoLibrary = optarg;
            break;
        case 0:
            if (long_opts[opt_idx].flag != 0) break;
        case 'h':
        default:
            LOG("-d/--demo-library [lib.so]  Demo library location");
            LOG("-h/--help                   Help");
            exit(c == 'h' ? 0 : 1);
        }
    }

    ENABLE_FE2();

    LOG("Initialising SDL");
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        LOG("ERROR: SDL initialisation failed: %s", SDL_GetError());
        exit(1);
    }
    atexit(SDL_Quit);
    if (TTF_Init() < 0) {
        LOG("ERROR: SDL-ttf initialisation failed: %s", TTF_GetError());
        exit(1);
    }
    atexit(TTF_Quit);
    SDL_WM_SetCaption("Demo player", "player");

    LOG("Initialising GL");
    pl_setVideoMode(800, 480);

    GLenum err = glewInit();
    if (err != GLEW_OK) {
        LOG("ERROR: GLEW initialisation failed: %s", glewGetErrorString(err));
        exit(1);
    }
    LOG("GLEW version: %s", glewGetString(GLEW_VERSION));

    LOG("OpenGL vendor: %s", glGetString(GL_VENDOR));
    LOG("OpenGL renderer: %s", glGetString(GL_RENDERER));
    LOG("OpenGL version: %s", glGetString(GL_VERSION));

#ifndef GLEW_NO_GLU
    LOG("GLU version: %s", gluGetString(GLU_VERSION));
#else
    LOG("GLU not present");
#endif
    CHECK_GL_ERROR("after OpenGL initialisation");
```

```cpp
    LOG("Setting OpenGL defaults");
    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.5);
    glDisable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LEQUAL);
    glDepthRange(0.f, 1.f);
    CHECK_GL_ERROR("after setting defaults");

    LOG("Loading demo library \"%s\"", demoLibrary);
    pl_loadDemoLib(demoLibrary);

    LOG("Initialising demo");
    demo_init(screen->w, screen->h);
    CHECK_GL_ERROR("after demo init");

    LOG("Beginning render loop");
    unsigned nFrames = 0, oldTicks = SDL_GetTicks();
    while (1) {
        if (!demo_prepareFrame()) {
            LOG("Demo finished");
            break;
        }
        CHECK_GL_ERROR("after frame preparation");

        bool done = false;
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            switch (event.type) {
            case SDL_MOUSEMOTION:
                if (demo_evtMouseMove) {
                    demo_evtMouseMove(event.motion.x,
                                      screen->h - event.motion.y - 1);
                }
                break;
            case SDL_MOUSEBUTTONDOWN:
            case SDL_MOUSEBUTTONUP:
                if (demo_evtMouseButton) {
                    demo_evtMouseButton(event.button.button,
                                        event.type == SDL_MOUSEBUTTONDOWN);
                }
                break;
            case SDL_QUIT:
                LOG("Window closed");
                done = true;
                break;
            }
        }
        if (done) break;

        Uint8 *keys = SDL_GetKeyState(NULL);
        if (keys[SDLK_ESCAPE]) {
            LOG("Escape pressed");
            break;
        }

        demo_drawFrame();
        CHECK_GL_ERROR("after frame rendering");
        SDL_GL_SwapBuffers();

        nFrames++;
        unsigned newTicks = SDL_GetTicks();
        if (newTicks - oldTicks > 2000) {
            INFO("%.1f fps (%d frames in %d ms)",
                 nFrames * 1000. / (newTicks - oldTicks),
                 nFrames, newTicks - oldTicks);
            oldTicks = newTicks;
            nFrames = 0;
        }
    }
    LOG("Exiting gracefully");
    return 0;
}
```

```
===
File: player/defsym.h (388 bytes)
===
```

```cpp
// hacky symbol magic
#ifndef defsym
#   error "need to define defsym to use defsym.h"
#endif

// define the symbols that are imported
defsym(true, demo_init,           void, unsigned, unsigned);
defsym(true, demo_prepareFrame, bool);
defsym(true, demo_drawFrame,    void);
defsym(false, demo_evtMouseButton,   void, uint8_t, bool);
defsym(false, demo_evtMouseMove,    void, int, int);
```

```
===
File: scenes/test_mesh/main.cpp (3623 bytes)
===
```

```cpp
#include "demo.h"
#include "mesh.h"
#include "text.h"
#include "pngwrite.h"

std::unique_ptr<Mesh> mesh;
ProgramMesh *program;
ProgramTextSDF *pTextSDF;
std::vector<std::unique_ptr<TextureTextSDF> > sdfNums;

void demo_init(unsigned, unsigned)
{
    program = new ProgramMeshDebugVizEdge;
    program->link();
    pTextSDF = new ProgramTextSDF(0);
    pTextSDF->link();

    /*
    mesh = Mesh::createRing(4, PI/4.f);
    mesh->splitVert(mesh->eEdge(0,0), mesh->eVertPrev(0,0));
    mesh->splitVert(mesh->eEdge(0,2), mesh->eVertPrev(0,2));
    mesh->splitVert(mesh->eEdge(0,4), mesh->eVertPrev(0,4));
    mesh->splitVert(mesh->eEdge(0,6), mesh->eVertPrev(0,6));
    mesh->verts[4].pos.x = 0;
    mesh->verts[5].pos.x = 0;
    mesh->verts[7].pos.y = 0;
    U32 edge = mesh->splitFace(mesh->eEdge(0,1), mesh->eEdge(0,5));
    mesh->splitVert(edge, mesh->eVertPrev(edge));
    mesh->debugOut();
    ASSERT(mesh->check());
    */

    mesh = Mesh::createGrid(3, 3);
    mesh->debugOut();
    ASSERT(mesh->check());

    program->updateVtxBuf(mesh);

    Font font("/usr/share/fonts/TTF/FreeSans.ttf", 300);
    for (int i = 0; i < 10; ++i) {
        sdfNums.push_back(std::unique_ptr<TextureTextSDF>(new TextureTextSDF(64, 64, font.draw(std::to_string(i)), 100)));
        sdfNums[i]->bind(0);
        sdfNums[i]->allocate();
    }
}

bool demo_prepareFrame()
{
    return true;
}

void demo_drawFrame()
{
    static unsigned i = 999;
    static unsigned u = 999;
    if (++u > 100) {
        u = 0;
        mesh->faces[i].mask = 0x00;
        if (++i >= mesh->faces.size()) i = 0;
        mesh->faces[i].mask = 0xff;
        program->updateVtxBuf(mesh, 0xff, 0x00);
    }

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    Mat4Stack ms;
    // camera-to-clip
    ms.push();
    ms.top = glm::perspective(35.0f, 1.f, 0.1f, 100.f);
    // world-to-camera
    ms.push();
    ms *= glm::translate(glm::mat4(), glm::vec3(0.f, 0.f, -10.f));

    static float t = 0.f;
    ms.push();
    //ms *= glm::rotate(glm::mat4(), t+=2.f, glm::vec3(0.f,1.f,0.f));
    program->use();
    program->setUniform("uTransform", ms.top);
    program->drawWire();
    program->unuse();

    glDisable(GL_DEPTH_TEST);
    //glEnable(GL_CULL_FACE);
    pTextSDF->use();
//   for (unsigned i = 0; i < mesh->faces.size(); ++i) {
        glm::vec3 faceNormal;
        const MeshFace &face = mesh->faces[i];
        for (int j = 0; j < face.count; ++j) {
            const MeshVert &vert = mesh->verts[face.verts[j]];
            pos += vert.pos;
            //faceNormal += vert.normal.
        }
        pos /= face.count;
        //faceNormal = glm::normalize(faceNormal);
        //faceNormal = glm::vec3(glm::inverse(glm::transpose(ms.top())) * glm::vec4(faceNormal, 1.f));
        /*
        if (glm::dot(faceNormal, glm::vec3(0.f, 0.f, -1.f)) < 0.f) {
            continue;
        }
        */
        ms.push();
        ms *= glm::translate(glm::mat4(), pos);
        ms *= glm::scale(glm::mat4(), glm::vec3(0.2f, -0.2f, 1.f));
        pTextSDF->setUniform("uTransform", ms.top);
        pTextSDF->setUniform("uTextParms.threshold", 0.5f);
        pTextSDF->setUniform("uTextParms.glow", false);
        sdfNums[i]->bind(0);
        pTextSDF->draw();
        ms.pop();
//}
    pTextSDF->unuse();
    glEnable(GL_DEPTH_TEST);
    //glDisable(GL_CULL_FACE);

    ms.pop();
    ms.pop();
    ms.pop();
}

void demo_evtMouseButton(uint8_t button, bool state)
{
}

void demo_evtMouseMove(int x, int y)
{
}
```

```
===
File: scenes/test_text/main.cpp (1788 bytes)
===
```

```cpp
#include "demo.h"
#include "mesh.h"
#include "text.h"
#include "pngwrite.h"
#include <SDL.h>

std::unique_ptr<Mesh> mesh;
ProgramTextSDF *program;
Font *font;
TextureTextSDF *textTex;

void demo_init(unsigned, unsigned)
{
    const char *str = "Hello world";
    program = new ProgramTextSDF(0/*, (ProgramTexturedQuad::vs, ProgramTexturedQuad::fs)*/);
    program->link();
    font = new Font("/usr/share/fonts/TTF/FreeSans.ttf", 600);
    auto size = font->measure(str);
    LOG("using size %dx%d", size.x, size.y);
    SDL_Surface *surf = font->draw(str);
    //PNGWrite(surf, "test.png");
    textTex = new TextureTextSDF(128, 128, surf, 200);
    surf = NULL;
    LOG("got size %dx%d", textTex->w, textTex->h);
    textTex->bind(0);
    textTex->allocate();
}

bool demo_prepareFrame()
{
    return true;
}

void demo_drawFrame()
{
    static float t = 0.f;
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    textTex->bind(0);

    program->use();
    glm::mat4 t_;
        A = glm::rotate(t, t*3.f, glm::vec3(1.f, 0.f, 0.f));
        B = glm::rotate(A, t*3.f, glm::vec3(0.f, 1.f, 0.f));
        C = glm::scale(B, glm::vec3(1.0f, -0.2f,1.f));
        C = glm::translate(C, glm::vec3(sin(t/30.f)*0.6f, sin(t/70.f)*0.3f, 0.f));
    program->setUniform("uTransform", B);
    program->setUniform("uTextParms.threshold", clamp((float) fabs(fmod(t/100.f,1.40f)-0.70f)*0.30f, 0.5f, 1.0f));
    program->setUniform("uTextParms.glowColor", glm::vec4(sin(t/100.f)/2+0.5, sin(t/100.f + 2*PI/3)/2+0.5, sin(t/100.f + 4*PI/3)/2+0.5, 1.f));
    t += 1.f;
    program->draw();
    program->unuse();
}

void demo_evtMouseMove(int x, int y)
{
}

void demo_evtMouseButton(uint8_t button, bool state)
{
}
```

```
===
File: scenes/test_rendertex/main.cpp (2364 bytes)
===
```

```cpp
#include "demo.h"
#include "mesh.h"
#include "text.h"
#include "pngwrite.h"
#include <SDL.h>
#include <unistd.h>

std::unique_ptr<Mesh> mesh;
ProgramTextSDF *program;
Font *font;
TextureTexture2D *renderTex;
FBO *renderTexFBO;

ProgramTexturedQuad *progMain;

void demo_init(unsigned, unsigned)
{
    const char *str = "Hello world";
    renderTexFBO = new FBO(128, 128, 0x0);
    renderTex = new TextureTexture2D(glm::uvec4<>(128, 128));
    renderTex->allocate();

    program = new ProgramTextSDF(0/*, (ProgramTexturedQuad::vs, ProgramTexturedQuad::fs)*/);
    program->link();
    progMain = new ProgramTexturedQuad(0);
    progMain->link();

    font = new Font("/usr/share/fonts/TTF/FreeSans.ttf", 600);
    auto size = font->measure(str);
    LOG("using size %dx%d", size.x, size.y);
    SDL_Surface *surf = font->draw(str);
    //PNGWrite(surf, "test.png");
    textTex = new TextureTextSDF(128, 128, surf, 200);
    surf = NULL;
    LOG("got size %dx%d", textTex->w, textTex->h);
    textTex->bind(0);
    textTex->allocate();
}

bool demo_prepareFrame()
{
    static int c = 0;
    if (++c == 200) {
        PNGWrite(renderTex, "text.png");
        return false;
    }
    return true;
}

void demo_drawFrame()
{
    static float t = 0.f;
    renderTexFBO->bindTexture("renderTex, 0);
    GLContext::get().bindFBO(*renderTexFBO);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    textTex->bind(0);

    program->use();
    program->setUniform("uTransform", glm::mat4());
    program->setUniform("threshold", 0.5f*clamp((float) fabs(fmod(t/100.f,1.40f)-0.70f)+0.30f, 0.5f, 1.0f)*/);
    program->setUniform("glowColor", glm::vec4(sin(t/100.f)/2+0.5, sin(t/100.f + 2*PI/3)/2+0.5, sin(t/100.f + 4*PI/3)/2+0.5, 1.f));
    t += 1.f;
    program->draw();
    program->unuse();

    GLContext::get().unbindFBO();

    renderTex->bind(0);
    renderTex->updateLocal();

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    progMain->use();
    renderTex->bind(0);
    progMain->setUniform("uTransform", glm::scale(glm::mat4(), glm::vec3(0.5f,-0.2f,1.0f)));
    progMain->draw();
    progMain->unuse();
}

/*
void demo_evtMouseMove(int x, int y)
{
}
void demo_evtMouseButton(uint8_t button, bool state)
{
}
*/
```

```
===
File: scenes/test_points/points.cpp (1526 bytes)
===
```

```cpp
#include "demo.h"

char *vertsPositions;

GLuint positionBufferObject;
GLuint vao, timeUniform, frame = 0;
Program *triangleProgram;

#define NPOINTS 5000000

void demo_init(unsigned w, unsigned h)
{
    // create shader program
    triangleProgram = new ProgramNew(
        Shader::FromFile(GL_VERTEX_SHADER, "data/points.vs"),
        Shader::FromFile(GL_FRAGMENT_SHADER, "data/points.fs")
    );

    // initialise vertex buffer
    vertsPositions = new char[NPOINTS];
    for (int i = 0; i < NPOINTS; ++i) {
        vertsPositions[i] = u;
    }

    glGenBuffers(1, &positionBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glBufferData(GL_ARRAY_BUFFER, NPOINTS, vertsPositions, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    glViewport(0, 0, (GLsizei) w, (GLsizei) h);

    glDisable(GL_PROGRAM_POINT_SIZE);
    glPointSize(2.0f);

    timeUniform = glGetUniformLocation(triangleProgram->id, "time");
}

bool demo_prepareFrame()
{
    return true;
}

void demo_drawFrame()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(triangleProgram->id);
    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 1, GL_UNSIGNED_BYTE, GL_TRUE, 0, 0);

    if (timeUniform != -1u) glUniform1f(timeUniform, frame++);

    glDrawArrays(GL_POINTS, 0, NPOINTS);

    glDisableVertexAttribArray(0);
    glUseProgram(0);
}
```

```
===
File: scenes/unittest/main.cpp (3369 bytes)
===
```

```cpp
#include "demo.h"
#include "mesh.h"
#include <random>
#include <functional>

std::unique_ptr<Mesh> mesh;

void demo_init(unsigned, unsigned)
{
    void _FAILTEST()
    {
        LOG("* UNIT TEST FAILED");
        abort();
    }

    template <typename T>
    void _TEST_EQ(T const &a, T const &b, const char *msg)
    {
        if (!(a == b)) {
            LOG("%s", msg);
            _FAILTEST();
        }
    }

    template <>
    void _TEST_EQ<float>(float const &a, float const &b, const char *msg)
    {
        if (fabs(a - b) > 1e-7) {
            LOG("%s  lhs = %f, rhs = %f", msg, (double) a, (double) b);
            _FAILTEST();
        }
    }

    #define TEST_EQ(expr, result) \
        _TEST_EQ(expr, result, #expr " != " #result)
    void demo_init(unsigned, unsigned)
    {
        std::normal_distribution<float> normDist;
        std::mt19937 engine; // Mersenne twister MT19937
        auto normRand = std::bind(normDist, engine);

        engine.seed(time(NULL));

        mesh = Mesh::createRing(4, PI/4.f);
        mesh->splitVert(mesh->eEdge(0,0), mesh->eVertPrev(0,0));
        mesh->splitVert(mesh->eEdge(0,2), mesh->eVertPrev(0,2));
        mesh->splitVert(mesh->eEdge(0,4), mesh->eVertPrev(0,4));
        mesh->splitVert(mesh->eEdge(0,6), mesh->eVertPrev(0,6));
        U32 edge = mesh->splitFace(mesh->eEdge(0,1), mesh->eEdge(0,5));
        ASSERT(edge/8 == 2 && mesh->eOpposite(edge)/8 == 0);
        ASSERT(mesh->check());

        TEST_EQ(lerp(0.f, 1.f, -1, 1), -1.f);
        TEST_EQ(lerp(0.f, 1.f, 0, 1), 0.f);
        TEST_EQ(lerp(0.f, 1.f, 1, 1), .5f);
        TEST_EQ(lerp(0.f, 1.f, 1, 1), 1.f);
        TEST_EQ(lerp(0.f, 1.f, 2, 1), 2.f);

        TEST_EQ(lerp(1.f, 0.f, -1, 1), 2.f);
        TEST_EQ(lerp(1.f, 0.f, 0, 1), 1.f);
        TEST_EQ(lerp(1.f, 0.f, 1, 2), .5f);
        TEST_EQ(lerp(1.f, 0.f, 1, 1), 0.f);
        TEST_EQ(lerp(1.f, 0.f, 2, 1), -1.f);

        TEST_EQ(lerp(Vec3(3.f,0.f,3.f)), Vec3((0.f,3.f,0.f), 2, 3),
            Vec3(1.f,2.f,1.f));

        {
            TEST_EQ(Mat2::identity().det(), 1.f);
            TEST_EQ(Mat2::identity()*4.f).det(), 16.f);

            Mat2 mr = {{{normRand(), normRand()},
                        {normRand(), normRand()}}};
            TEST_EQ(mr*Mat2::identity(), mr);
            TEST_EQ(Mat2::identity()*mr, mr);
            TEST_EQ(mr.transpose().transpose(), mr);
            TEST_EQ(mr.transpose().det(), mr.det());
        }

        {
            Mat3 mr = {{{normRand(), normRand(), normRand()},
                        {normRand(), normRand(), normRand()},
                        {normRand(), normRand(), normRand()}}};
            TEST_EQ(Mat3::identity().det(), 1.f);
            TEST_EQ(Mat3::identity()*4.f).det(), 64.f);

            Mat3 ma({{{-2.f,-1.f,2.f},
                      {2.f,1.f,-1.f},
                      {2.f,3.f,-1.f}}});

            TEST_EQ(ma[0][0], -2.f); TEST_EQ(ma[0][1], -1.f); TEST_EQ(ma[0][2], 2.f);
            TEST_EQ(ma[1][0], 2.f); TEST_EQ(ma[1][1], 1.f); TEST_EQ(ma[1][2], -1.f);
            TEST_EQ(ma[2][0], 3.f); TEST_EQ(ma[2][1], 3.f); TEST_EQ(ma[2][2], -1.f);
            TEST_EQ(ma[0][0] * (ma[1][1]*ma[2][2] - ma[2][1]*ma[1][2]), 2.f);
            TEST_EQ(ma[1][0] * (ma[2][1]*ma[0][2] - ma[0][1]*ma[2][2]), 10.f);
            TEST_EQ(ma[2][0] * (ma[0][1]*ma[1][2] - ma[1][1]*ma[0][2]), -6.f);
            TEST_EQ(ma.det(), 6.f);
            TEST_EQ(ma.transpose().det(), 6.f);
            TEST_EQ(ma.transpose().transpose(), ma);
        }
    }

    bool demo_prepareFrame()
    {
        return false;
    }

    void demo_drawFrame()
    {
    }

    void demo_evtMouseMove(int x, int y)
    {
    }

    void demo_evtMouseButton(uint8_t button, bool state)
    {
    }
```

```
===
File: scenes/test_sdf/impostor.cpp (1578 bytes)
===
```

```cpp
#include "demo.h"

const float vertsPositions[] = {
    -1.0f, -1.0f, 0.0f, 1.0f,
```

```
            -1.0f, +1.0f, 0.0f, 1.0f,
            +1.0f, +1.0f, 0.0f, 1.0f,
            +1.0f, -1.0f, 0.0f, 1.0f
};

GLuint positionBufferObject;
GLuint vao;
Program *triangleProgram;
GLuint timeUniform, frame = 0;

void demo_init(unsigned w, unsigned h)
{
    // create shader program
    triangleProgram = new ProgramRaw({
        Shader::FromFile(GL_VERTEX_SHADER, "data/impostor.vs"),
        Shader::FromFile(GL_FRAGMENT_SHADER, "data/impostor.fs")
    });
    triangleProgram->link();

    // initialise vertex buffer

    glGenBuffers(1, &positionBufferObject);
    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    CHECK_GL_ERROR("1");

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);
    CHECK_GL_ERROR("2");

    glViewport(0, 0, (GLsizei) w, (GLsizei) h);

    timeUniform = glGetUniformLocation(triangleProgram->id, "time");
    CHECK_GL_ERROR("3");
}

bool demo_prepareFrame()
{
    return true;
}

void demo_drawFrame()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(triangleProgram->id);

    if (timeUniform != -1u) glUniform1f(timeUniform, frame++);

    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glDisableVertexAttribArray(0);
    glUseProgram(0);
}


===
File: scenes/test_triangle/triangle.cpp (1281 bytes)
===

#include "demo.h"

const float vertexPositions[] = {
    0.75f, 0.75f, 0.0f, 1.0f,
    0.75f, -0.75f, 0.0f, 1.0f,
    -0.75f, -0.75f, 0.0f, 1.0f,
};

GLuint positionBufferObject;
GLuint vao;
Program *triangleProgram;

void demo_init(unsigned w, unsigned h)
{

    // create shader program

    triangleProgram = new ProgramRaw({
        Shader::FromFile(GL_VERTEX_SHADER, "data/triangle.vs"),
        Shader::FromFile(GL_FRAGMENT_SHADER, "data/triangle.fs")
    });

    // initialise vertex buffer

    glGenBuffers(1, &positionBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}

bool demo_prepareFrame()
{
    return true;
}

void demo_drawFrame()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(triangleProgram->id);

    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glUseProgram(0);
}


===
File: scenes/test_wave/main.cpp (7251 bytes)
===

#include "demo.h"
#include "wave.h"
#include "program.h"
#include "imgwrite.h"
#include "text.h"
#include <cmath>
#include <array>

const std::shared_ptr<Shader> wavefs =
    Shader::inline(GL_FRAGMENT_SHADER, R"(
        #version 130
        in vec2 vTexCoord;
        uniform mat4 uTransform;
        uniform sampler2D uTex;
        const float delta = 0.005f;
        uniform vec3 lightray = vec3(1.f, 1.f, 1.f);
        uniform int shadeStyle = 0;
        const float M_PI = 3.141592653589793f;

        vec3 getSurf(vec2 pos) {
            return vec3(uTransform*vec4(pos, pow(texture(uTex, pos).r, 1.0f), 1.f));
        }

        void main() {
            vec3 normal = normalize(cross(
                getSurf(vTexCoord+vec2(delta, 0.f))-getSurf(vTexCoord+vec2(-delta, 0.f)),
                getSurf(vTexCoord+vec2(0.f, delta))-getSurf(vTexCoord+vec2(0.f, -delta))));
            vec3 lightdir = normalize(lightray);
            //float angle = clamp(acos(dot(normal, lightdir)), 0.f, M_PI);
            float angle = clamp(dot(normal, lightdir), 0.f, 1);
            float value = clamp(angle/M_PI, 0.f, 1.f);
            vec3 halfang = normalize(lightdir + vec3(0.f, 0.f, -1.f));
            float specval = clamp(dot(normal, halfang), 0.f, 1.f);
            specval = angle != 0.0f ? specval : 0.0f;

            vec3 ambient = vec3(1.0f, 1.0f, 1.0f) * 0.1f;
            vec3 specular = 5.0f*pow(specval, 5.0f) * vec3(0.78f, 0.90f, 1.0f);
            vec3 diffuse = value * vec3(0.38f, 0.45f, 0.9f);

            switch (shadeStyle) {
            case 0:
                gl_FragColor = vec4(ambient + specular + diffuse, 1.f);
                break;
            case 1:
                gl_FragColor = vec4(normal*2, 1.f);
                break;
            case 2:
                gl_FragColor = vec4(specular, 1.f);
                break;
            case 3:
                gl_FragColor = vec4(diffuse, 1.f);
                break;
            }
            //gl_FragColor = vec4(normal*100, 1.f);
        }
    )", "custom wavefs");

Wave *wave;
TextureWave *waveTex;
ProgramTexQuadImp *program;

int tick = 0;
int stepstep = -1;
int shadeStyle = 0;

Font *font;
const char *texts[] = {
    "Moving waves",
    "Double slit experiment",
    "Water drops",
    "Blinn-Phong shading",
    "Normals",
    "Specular",
    "Diffuse"
};
static const int texts_shadeOff = 3;
static const int nShadeStyles = 4;
static constexpr int ntexts = sizeof(texts) / sizeof(texts[0]);
TextureTextSDF *textTexs[ntexts];
ProgramTextSDF *programText;

void nextstepstep()
{
    tick = 0;
    stepstep++;
    if (stepstep > 2) stepstep = 0;
    wave->reset();
    switch (stepstep) {
    case 0:
        break;
    case 1:
        for (int i = 0; i < 128-16; ++i) {
            wave->Wset(128, i, true);
        }
        for (int i = 128-12; i < 128+12; ++i) {
            wave->Wset(128, i, true);
        }
        for (int i = 128+16; i < 256; ++i) {
            wave->Wset(128, i, true);
        }
        break;
    case 2:
        for (int i = 30; i < 226; ++i) {
            wave->Wset( 30, i, true);
            wave->Wset(225, i, true);
            wave->Wset(i,  30, true);
            wave->Wset(i, 225, true);
        }
        break;
    }
}

void demo_init(unsigned, unsigned)
{
    wave = new Wave(256, 256);
    nextstepstep();
    waveTex = new TextureWave(*wave);
    waveTex->bind(0);
    waveTex->allocate();
    program = new ProgramTexQuadImp(0, {ProgramTexQuadImp::vs, wavefs});
    program->link();

    programText = new ProgramTextSDF(1/*, {ProgramTexturedQuad::vs, ProgramTexturedQuad::fs}*/);
    programText->link();
    font = new Font("/usr/share/fonts/TTF/FreeSans.ttf", 200);
    for (int i = 0; i < ntexts; ++i) {
        auto size = font->measure(texts[i]);
        LOG("using size %d×%d", size.x, size.y);
        SDL_Surface *surf = font->draw(texts[i]);
        textTexs[i] = new TextureTextSDF(128, 32, surf, 90);
        surf = NULL;
        textTexs[i]->bind(0);
        textTexs[i]->allocate();
    }
}

bool demo_prepareFrame()
{

    /*
    if (++c == 200) {
        PNGWrite(waveTex, "test.png");
        return false;
    }
    */
    switch (stepstep) {
    case 0:
        for (int i = 0; i < 10; ++i) {
            wave->D[30+(tick/50)%196,200] = cosf(tick * 0.05f);
            wave->D[100,30+(tick/20)%196] = cosf(tick * 0.05f);
            tick++; wave->update();
        }
        //if (tick > 5000) nextstepstep();
        return true;
    case 1:
        for (int t = 0; t < 10; ++t) {
            for (int i = 51; i < 205; ++i) {
                wave->D(51,i) = cosf(tick * 0.05f);
            }
            tick++; wave->update();
        }
        //if (tick > 5000) nextstepstep();
        return true;
    case 2:
    {
        int iters = 6;
        /*if (tick >= 200 && tick <= 600) {
            iters = 2;
        }*/
        for (int t = 0; t < iters; ++t) {
            if (tick % 500 == 30) {
                wave->D(51,136) = cosf((tick*500)*PI/60.f);
            }
            if ((tick%250) % 500 == 30) {
                wave->D(141,170) = cosf((tick+250\500)*PI/60.f);
```

(second column)

```
                }
                tick++; wave->update();
            }
            //if (tick > 5000) nextstepstep();
            return true;
        default:
            return false;
        }
    }

    void demo_drawFrame()
    {
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        glClearDepth(1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        waveTex->bind(0);
        waveTex->update();

        program->use();
        static float time = 0.f;
        //program->setUniform("uTransform", glm::rotate(glm::mat4(), time+=0.1f, glm::vec3(0.1f, 0.3f, 1.f)));
        //program->setUniform("uTransform", glm::mat4());
        program->setUniform("uTexScale", 0.7f);
        program->setUniform("shadeStyle", shadeStyle);
        program->setUniform("lightray", glm::vec3(cosf(time/100)*2.f-1.f, sinf(time/100)*2.f-1.f, 1.f));
        program->draw();
        program->unuse();

        programText->use();
        programText->setUniform("uTextParms.threshold", 0.5f);
        programText->setUniform("uTextParms.glowColor", glm::vec4(0.f, 0.f, 0.f, 1.f));
        programText->setUniform("uTextParms.glow", true);

        programText->setUniform("uTransform",
            glm::scale(
                glm::translate(
                    glm::mat4(),
                    glm::vec3(0.4f, -0.8f, 0.0f)
                ),
                glm::vec3(0.03f*strlen(texts[stepstep]), -0.2f, 1.f)*0.3f
            ));
        textTexs[stepstep]->bind(1);
        programText->draw();

        programText->setUniform("uTransform",
            glm::scale(
                glm::translate(
                    glm::mat4(),
                    glm::vec3(0.4f, -0.8f, 0.0f)
                ),
                glm::vec3(0.03f*strlen(texts[shadeStyle + texts_shadeOff]), -0.2f, 1.f)*0.3f
            ));
        textTexs[shadeStyle + texts_shadeOff]->bind(1);
        programText->draw();

        programText->unuse();

        time += 1.f;
    }

    void demo_evtMouseButton(uint8_t button, bool state)
    {
        if (state == false) {
            switch (button) {
            case 1:
                nextstepstep();
                break;
            case 3:
                shadeStyle = (shadeStyle+1) % nShadeStyles;
                break;
            }
        }
    }
```