Criação das telas

1)VTK

A VTK é um toolkit de 3d que, entre outras coisas, abstrai as complexidades de criar e manter contextos 3d, fornecendo uma maneira portável de faze-lo.

* 1. Sistema de renderização da VTK: ele tem os seguintes componentes: actors, renderers, câmeras, luzes, renderwindows e interactors.

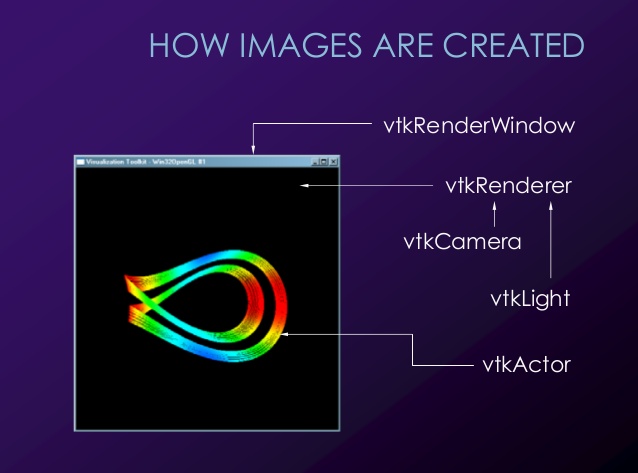
Os actors são os responsáveis pela renderização propriamente dita, tendo posição no espaço, transformações e rotinas que são invocadas quando o renderer percorre a lista de actors os renderizando. Em geral cada actor delega o processo de renderização para mappers, como será visto mais à frente, que são as entidades responsáveis por estabelecer a ligação entre a representação gráfica e os dados sendo representados. As várias classes descendentes do actor lidam com tipos específicos de actors, com suas necessidades especiais, como a diferença entre renderizar polígonos e renderizar volumes.

Os renderers são abstrações dos contextos gráficos, como o contexto da opengl, encapsulando coisas como, no caso do opengl, fazer o contexto ser o contexto corrente, antialiasing, renderização estéreo (para óculos 3d). Um exemplo desse encapsulamento é a quase transparência no uso do opengl 3.2 e opengl 1.5 legado, escolhido durante a compilação da VTK. Se o programador não fizer uso de nada baixo-nível o seu código deve compilar sem nenhuma alteração depois de trocar qual versão do opengl será usada. As várias classes descendentes definem os renderers apropriados para cada biblioteca gráfica.

As câmeras são contidas nos renderers e são basicamente mecanismos de manipulação da matriz ViewProjection usada na renderização, com a matriz model sendo provida pelo actor.

As luzes, por sua vez abstraem as funcionalidades de iluminação, como o tipo de iluminação e posicionamento da luz.

A renderwindow é a cola do renderer com o sistema operacional. Cada sistema operacional tem a sua maneira de criar e manter contextos gráficos e de lidar com eventos. A renderwindow abstrai esses detalhes e provê um mecanismo unificado. As classes derivadas da renderwindow lidam com as várias formas de ligação entre o contexto gráfico e o sistema operacional.



Slide de <https://www.slideshare.net/waldyrious/introduction-to-vtk> .

* 1. Smart Pointers: os smart pointers, no c++ servem para simplificar o manejamento da memória. Em c++ isso é necessário pois a linguagem/ambiente de execução não provê nenhum mecanismo de garbage collection. No c++ os ponteiros podem estar expostos, onde a gerência da memória é toda responsabilidade do programador, ou encapsulados em smart pointers, que abstraem essa responsabilidade.

No nosso caso há 3 sistemas diferentes de smart pointers: os smart pointers da ITK, os smart pointers da VTK e os da STL. Os da ITK, da VTK e os shared\_ptr da STL funcionam de maneira similar, por contagem de referência. Cada vez que são atribuídos a uma variável do tipo smart pointer eles incrementam de valor. Toda vez que uma variável do tipo smart pointer sai do escopo o contador decrementa. Se o contador chega à zero aquilo que o smart pointer guarda é deletado.

Tal mágica funciona porque os smart pointers são objetos. Quando um objeto criado no stack no c++ sai do escopo seu destrutor é invocado automaticamente. No destrutor dessa instância o contador, que foi aumentado no construtor, é decrementado.

Em

void foo()

{

std::shared\_ptr<myType> p = std::make\_shared<myType>();

//... faz algo

}

A variável local p é um smart pointer do tipo myType. Quando foo() acabar e as variáveis locais de foo forem eliminadas o construtor de p será invocado. Se não houver nenhum outro uso do objeto gerenciado por p o objeto terá seu destrutor invocado.

Os vários tipos de smartpointers permitem acesso ao ponteiro de baixo nível, “nu”, necessário para muitas APIs que não conhecem smartpointers. Se o programador deletar manualmente esse ponteiro nu ele terá estragado o ponteiro e vai causar um erro mais na frente porque os smartpointers não tem controle algum com o que é feito com o ponteiro nu caso o programador decida usá-lo diretamente.

Além disso os vários tipos de smartpointer são incompatíveis entre si. Se um bloco de memória está gerenciado por um smartpointer da STL ele não pode ser gerenciado por outro tipo de smartpointer, digamos, o da ITK, porque cada um terá sua contagem de referências separadas e quando um deles chegar a zero o bloco será desalocado, mesmo que esteja em uso em outro lugar. Porém um objeto que tenha como membros smartpointers pode ter sua memória gerenciada por um tipo de smartpointer enquanto seus membros tem por outro tipo.

Embora eu tenha falado muito de objetos, as restrições a objetos só existem, dentre os smartpointers que serão usados aqui, nos smartpointers da ITK e VTK, que só aceitam objetos derivados de certos tipos básicos existentes nessas bibliotecas. Os smartpointers da STL aceitam qualquer tipo de bloco de memória.

2)A classe Tela

Essa classe encapsulará os vários objetos necessários para a criação de uma tela.

2.1)Descrição: A tela mostrará a renderização e receberá interação do usuário. Ela tem um método de renderização e um método de iniciar o loop de eventos. Dentro dela tem um renderer, um renderwindow e um interactor.

2.2)O código

Tela.h

#pragma once

#include <vtkRenderer.h>

#include <vtkRenderWindow.h>

#include <vtkRenderWindowInteractor.h>

#include <vtkSmartPointer.h>

/\*A minha tela.

\* Forma de usar:

\* 1)Construir o objeto

\* 2)Invocar Render(). A tela, caso não tenha sido criada, será criada nesse momento.

\* 3)Quando estiver pronto para começar o event loop, invocar StartEventLoop.

\*/

class Tela

{

private:

vtkSmartPointer<vtkRenderWindow> renderWindow;

vtkSmartPointer<vtkRenderer> renderer;

vtkSmartPointer<vtkRenderWindowInteractor> interactor;

public:

/\*Construtor - Cria a tela, mas para ela começar a captar os eventos, ativar o loop de

\* eventos, a ultima tela a ser criada terá que invocar StartEventLoop();\*/

Tela();

/\*Renderiza a tela.\*/

void Render();

/\*Começa o loop de eventos. O thread principal irá entrar nessa função e só voltará quando

\* a janela for fechada, então deve ser a última coisa a ser invocada porque o que vier depois

\* dela só será chamado quando a janela fechar.\*/

void StartEventLoop();

void CriarObjetoDeTeste();

};

Tela.cpp

#include "Tela.h"

#include "vtkProperty.h"

#include "vtkCamera.h"

#include "vtkCubeSource.h"

#include "vtkPolyDataMapper.h"

#include "vtkActor.h"

Tela::Tela()

{

renderer = vtkSmartPointer<vtkRenderer>::New();

renderWindow = vtkSmartPointer<vtkRenderWindow>::New();

renderWindow->SetSize(300, 300);

renderWindow->AddRenderer(renderer);

interactor = vtkSmartPointer<vtkRenderWindowInteractor>::New();

interactor->SetRenderWindow(renderWindow);

}

void Tela::Render()

{

renderWindow->Render();

}

void Tela::StartEventLoop()

{

interactor->Start();

}

void Tela::CriarObjetoDeTeste()

{

vtkSmartPointer<vtkCubeSource> testeCubeSource = vtkSmartPointer<vtkCubeSource>::New();

testeCubeSource->SetCenter(0, 0, 0);

testeCubeSource->SetXLength(1);

testeCubeSource->SetYLength(1);

testeCubeSource->SetZLength(1);

vtkSmartPointer<vtkPolyDataMapper> testeCubeMapper = vtkSmartPointer<vtkPolyDataMapper>::New();

testeCubeMapper->SetInputConnection(testeCubeSource->GetOutputPort());

vtkSmartPointer<vtkActor> testeCubeActor = vtkSmartPointer<vtkActor>::New();

testeCubeActor->SetMapper(testeCubeMapper);

testeCubeActor->GetProperty()->SetColor(1.0000, 0.3882, 0.2784);

testeCubeActor->RotateX(30.0);

testeCubeActor->RotateY(-45.0);

renderer->AddActor(testeCubeActor);

renderer->ResetCamera();

}

No constructor eu crio os objetos necessários para tela vazia: um renderer, um renderwindow e um interactor. Não escolhi nenhum tipo especifico, então a VTK usará a melhor implementação para o sistema operacional em que eu estou (no meu caso, Windows, usa herdeiros apropriados para execução no Windows usando opengl). Eu crio os objetos e os adiciono uns aos outros. Os objetos são smartpointers com escopo de objeto. Quando o destrutor do objeto for invocado os smartpointers terão seus destrutores invocados que por sua vez reduzirão os contadores dos objetos por eles guardados e se os contadores chegarem a zero, os objetos guardados nos smartpointers serão deletados, coletando assim os recursos. Os recursos são adquiridos no construtor porque no C++ recomenda-se o uso do padrão RAII – Resource Aquisition is Initialization.

Em Render simplesmente se invoca o renderizador DA JANELA. Um erro comum é invocar o método render() do renderer. Se isso for feito nem sempre vai renderizar. O processo de renderização começa na janela, que fará as coisas necessárias a nível de sistema operacional, para se começar a renderizar (ex.: invocar wglMakeCurrent()).

StartEventLoop inicia o loop de eventos, invocando o método start do interactor. Esse método não retorna enquanto o loop de eventos estiver ativo pois esse loop é um loop infinito.

CriarObjetoDeTeste é um método que existe temporariamente para mostrar que a tela estará funcionando e seu funcionamento será explicado mais à frente

Observação: depois de criar novos arquivos no diretório de sources é obrigatório rodar o cmake novamente para atualizar o makefile do projeto.

O segmentator.cpp fica da seguinte forma:

#include "Tela.h"

#include <memory>

using namespace std;

int main(int argc, char\*\* argv)

{

shared\_ptr<Tela> tela00, tela01;

tela00 = make\_shared<Tela>();

tela00->CriarObjetoDeTeste();

tela00->Render();

tela01 = make\_shared<Tela>();

tela01->CriarObjetoDeTeste();

tela01->Render();

tela01->StartEventLoop();

return EXIT\_SUCCESS;

}

Ele cria dois objetos do tipo Tela, encapsulados em smartpointers da STL. Como dito acima, esse tipo de combinação, um objeto com smartpointers de um tipo sendo gerenciado por um smartpointer de um tipo diferente é válido.

3) O objeto de teste

Os objetos na VTK são gerados por pipelines: sequencias de objetos que fazem processamentos em cima de entradas e geram resultados para as próximas etapas

3.1) A Pipeline: Os objetos de pipeline da VTK normalmente recebem entradas e geram resultados.

Há três tipos de objetos numa pipeline da VTK: Sources, Filters e Mappers. Sources fornecem dados para aqueles que vierem depois dela mas elas próprias não recebem dados de nenhum objeto. Um bom exemplo de source são os carregadores de arquivos. Filters recebem dados, realizam (ou não) processamento sobre eles e os disponibilizam para aqueles que vierem depois. Um bom exemplo de filter são filtros de suavização gaussiana, que recebem uma imagem e a modificam, retornando a imagem modificada. E mappers por sua vez são finais de pipeline, eles recebem dados e os mapeiam para representações, como por exemplo mapear vetores para pontos no espaço. Eles não fornecem resultados para filtros.

Os resultados dos processamentos são armazenados em cache nos objetos ao menos que se especifique o contrário. Quando atributos de um elemento da pipeline muda ele requisita atualizações de todos os que vieram antes dele na pipeline. Aqueles que por sua vez tiverem sido modificados requisitarão mudanças também, até chegar no começo. Quando chega nesse ponto, o processamento começa, indo na direção inversa da requisição.

Exemplo: uma pipeline com uma source, dois filtros e um mapper. Quando o mapper for usado, digamos, para renderizar algo na tela, ele pedirá os dados do filter 2, que pedirá os dados do filter 1, que pedirá os dados do source. Se nada tiver sido modificado as versões em cache da ultima modificação serão usadas. Se tiver ocorrido modificações cada objeto que sofreu modificação realizará de nosso seu processamento e guardará o resultado. A ordem de processamento será do source para o mapper.

3.2) Exemplo: Para por um cubo na tela eu crio uma source de cubo, passo esse resultado para um mapper de polígonos e o polígono para um actor, que por sua vez é posto em um renderer. Quando o renderer se renderizar ele requisitará que cada actor se renderize. O actor pedirá para o mapper fazer esse renderzação, o mapper pedirá para os filtros/sources que o alimentam para realizarem o processamento caso necessário, indo assim até o source.

vtkSmartPointer<vtkCubeSource> testeCubeSource = vtkSmartPointer<vtkCubeSource>::New();

testeCubeSource->SetCenter(0, 0, 0);

testeCubeSource->SetXLength(1);

testeCubeSource->SetYLength(1);

testeCubeSource->SetZLength(1);

vtkSmartPointer<vtkPolyDataMapper> testeCubeMapper = vtkSmartPointer<vtkPolyDataMapper>::New();

testeCubeMapper->SetInputConnection(testeCubeSource->GetOutputPort());

vtkSmartPointer<vtkActor> testeCubeActor = vtkSmartPointer<vtkActor>::New();

testeCubeActor->SetMapper(testeCubeMapper);

testeCubeActor->GetProperty()->SetColor(1.0000, 0.3882, 0.2784);

testeCubeActor->RotateX(30.0);

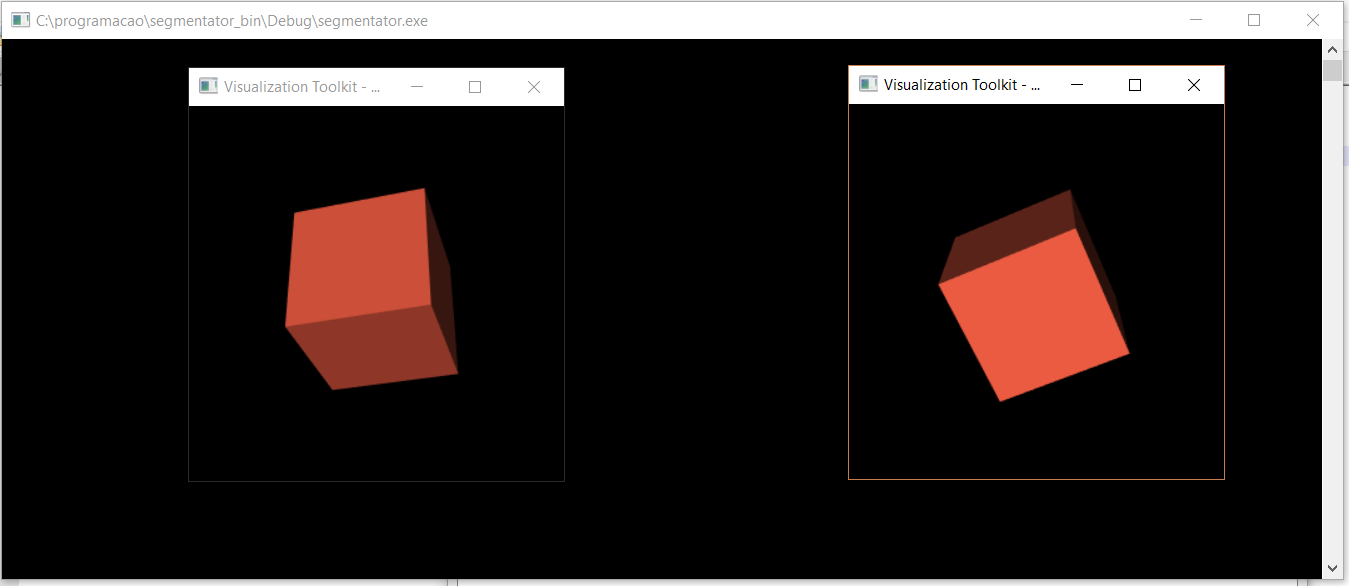
testeCubeActor->RotateY(-45.0);

renderer->AddActor(testeCubeActor);

renderer->ResetCamera();

Os objetos criados nesse método não serão deletados no final do método pois testeCubeSource é referenciado por testeCubeMapper, testeCubeMapper é referenciado por testeCubeActor, que por sua vez é referenciado por renderer. Há pelo menos uma referência para cada objeto e tais referências existirão pelo menos enquanto o renderer existir, nesse caso nenhum objeto será deletado.

4) Resultado final:



O resultado final são dois cubos, um em cada tela, que podem ser operados independentemente. Caso a tela 1 seja fechada o loop de eventos continua a ser executado, mas se a tela dois for fechada o loop para porque o interactor que começou o loop de eventos foi o da tela 2.

Criar as duas telas em threads separados não é uma boa ideia. A VTK, pelo menos no seu sistema de janela, não é thread-safe e nunca consegui fazer uma versão que use múltiplas janelas, cada uma em thread, funcionar para coisas não-triviais sem que exceções fossem disparadas. O mesmo vale para a criação de REPLs (Read, Eval, Print, Loop) onde o loop do console ocorre em um thread e o loop da cena 3d na tela em outro.