

ВВЕДЕНИЕ (1)

ЗАПУСК И НАСТРОЙКА GEANT4

- Запуск простейшего примера(1)
- RunManager(2)
- VisManager(2)
- UImanager(2)
- Рабочее окно визуального интерактивного режима(2)

ГЕОМЕТРИЯ В GEANT4

- Единицы измерения(2.5)
- Материалы. Создание Материалов(3)
- База Материалов NIST(3)
- Основы реализации геометрии(4)
- Формы(4)
- Логические объемы(4)
- Физические объемы(4)
- Контейнеры-хранилища(4)
- Логические операции над формами(4)
- Контейнеры для логических объемов(4)

ФИЗИЧЕСКИЕ ПРОЦЕССЫ В GEANT4

- Использование готовых физических моделей (5)
- Создание физического процесса (5)
- Сборка физической модели (5)

КЛАССЫ ДЕЙСТВИЙ

- Генерация первичных событий(6)
- Использование функциональной зависимости в генераторе первичного излучения(6)

ЦИКЛ ОБРАБОТКИ СОБЫТИЙ

- Шаги(7)
- Треки(7)
- Стек при работе с треками(7)
- События(7)
- Запуски(7)
- Сбор информации с шагов в запуске(7)
- Многопоточные приложения в Geant4(8)

ИНТЕРАКТИВНЫЙ РЕЖИМ И КОМАНДЫ

- Встроенные команды(9)
- Интерактивный режим(9)
- Вызовы команд из кода(9)
- Макрос-файлы(9)
- Создание пользовательских команд(9)
- Пользовательские команды, связанные с геометрией(9)

ПРИЛОЖЕНИЕ

1. Установка Geant4 под Linux(10)
2. Установка Geant4 под Windows(10. Отсутствует)

1. Введение

Современные эксперименты в ядерной физике испытывают множество проблем в процессе разработки комплекса программных средств и приложений. Актуальной становится постоянно растущая необходимость в исчерпывающем и точном моделировании регистрации частиц. Цель данного пособия, изучить способы применения современного инструментария математического моделирования Geant4 в решении задач атомной промышленности и ядерной энергетики реально возникающих задачах атомной промышленности и ядерной энергетики.

Название Geant4 произошло от английского Geometry And Tracking (геометрия и трекинг). Geant4 - это инструментарий для моделирования прохождения элементарных частиц через вещество на основе методов Монте-Карло. В сердце данного пакета богатый набор физических моделей, содержащий информацию о взаимодействии частиц с материалами в широком диапазоне энергий. Данные для моделей получены из огромного количества источников по всему миру, и в этом отношении Geant4 представляет собой беспрецедентное хранилище информации, включающее в себя значительную часть всего, что известно о взаимодействиях частиц. Более того, из года в год он продолжает наполняться, разрабатываться и расширяться.

Процесс разработки Geant4 начался в далеком 1993 году в результате двух независимых исследований, проводимых CERN и KEK. Обе группы занимались изучением того, как современные компьютерные технологии могут быть использованы для улучшения уже существующих пакетов математического моделирования, вроде Geant3, являющегося на тот момент эталоном, а также источником идей и ценного опыта. Объединение результатов данных исследований было представлено в CERN Detector Research and Development Committee (DRDC) с целью создания системы моделирования на основе объектно-ориентированных технологий. Так был

создан проект RD44, ставший результатом сотрудничества ученых и инженеров десятков экспериментов, проводимых в Европе, Канаде, России, США и Японии. В декабре 1998 год состоялся релиз первой версии продукта. Вскоре, в январе 1999 года было создано сообщество, целью которого стала дальнейшая разработка и совершенствование продукта. Сам же продукт получил название Geant4.

В рамках данного пособия используется семантика языка C++. Пособие подразумевает, что читатель разбирается в основных принципах объектно-ориентированного программирования (ООП) и, в частности, возможностях и особенностях языка C++. Кроме того, считается, что читатели понимают основы теории вероятности, а также владеют методами математической статистики.

В пособии будут рассмотрены основы инструментария Geant4 на примере простейших учебных задач, в основу которых легли научные работы кафедры № 24 «Прикладная ядерная физика» НИЯУ МИФИ.

В завершении книги читатель найдет подробное руководство по настройке и установке Geant4 на различных операционных системах.

Предполагается, что данное пособие будет полезно для магистров направления обучения 14.04.02 «Ядерная физика и технологии», изучающих курс «Математическое моделирование: Geant4».

1. Запуск простейшего примера

Первое, с чего стоит начать после завершения установки (а заодно и убедиться, что все настроено и работает правильно) - с запуска одного из встроенных примеров. В комплекте с установленной средой моделирования пользователь получает 3 группы примеров: `basic`, `extended` и `advanced`. Начать, пожалуй, стоит с самого простого примера B1, расположенного в каталоге `basic`. Данный каталог пользователь может найти в директории с установленным Geant4. Путь будет выглядеть следующим образом:

`${Geant4_install}[/share/Geant4-x/examples/basic/B1`

Примечание: в рамках данного пособия во всех примерах используется среда IDE Clion от JetBrains, поэтому запуск будет рассмотрен в реалиях этой среды (подробнее в Приложении Б):

Шаг 1. Открытие примера

Откроем проект в Clion:

File > Open...

И укажем **путь** (пусть) к папке с примером.

Шаг 2. Настройка Cmake

После запуска в консоли может отобразиться сообщение об ошибке, в котором сказано, что не удалось найти расположение файлов Geant4:

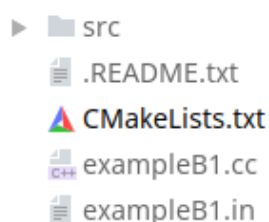
```
CMake Error at CMakeLists.txt:15 (find_package):
  By not providing "FindGeant4.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Geant4", but
  CMake did not find one.

Could not find a package configuration file provided by "Geant4" with any
of the following names:

  Geant4Config.cmake
  geant4-config.cmake

Add the installation prefix of "Geant4" to CMAKE_PREFIX_PATH or set
"Geant4_DIR" to a directory containing one of the above files.  If "Geant4"
provides a separate development package or SDK, be sure it has been
installed.
```

Эту ошибку легко исправить: следует открыть в корне проекта файл CMakeList.txt



```
► src
  .README.txt
  CMakeLists.txt
  exampleB1.cc
  exampleB1.in
```

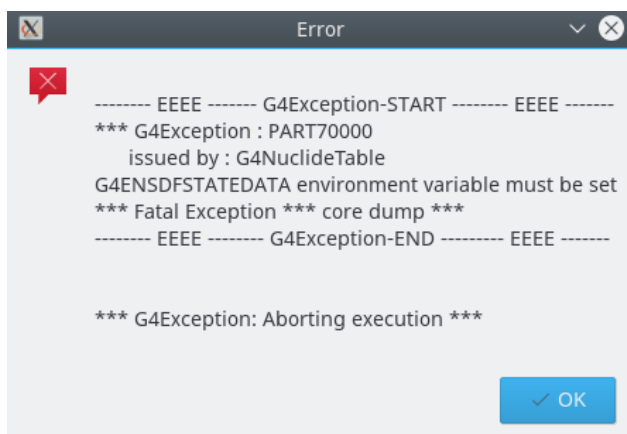
`${PATH_TO_GEANT4}`

Затем следует добавить строчку, которая сообщит сборщику путь к Geant4:

Затем следует перезагрузить проект, и ошибка исчезнет.

Шаг 3. Настройка библиотек Geant4

Если собрать проект, при запуске пользователь увидит следующее сообщение:

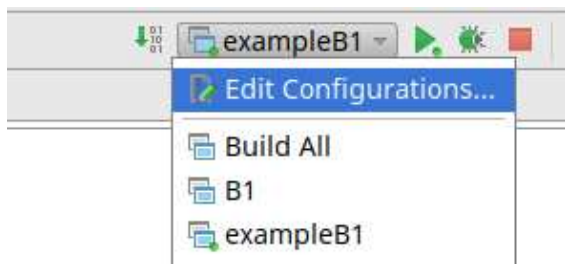


В данном сообщении сказано, что не указаны пути к базам данных Geant4. Найти полный список требований для каждого примера, а также информацию о том, что в нём реализовано, можно получить в приложенном файле README.txt

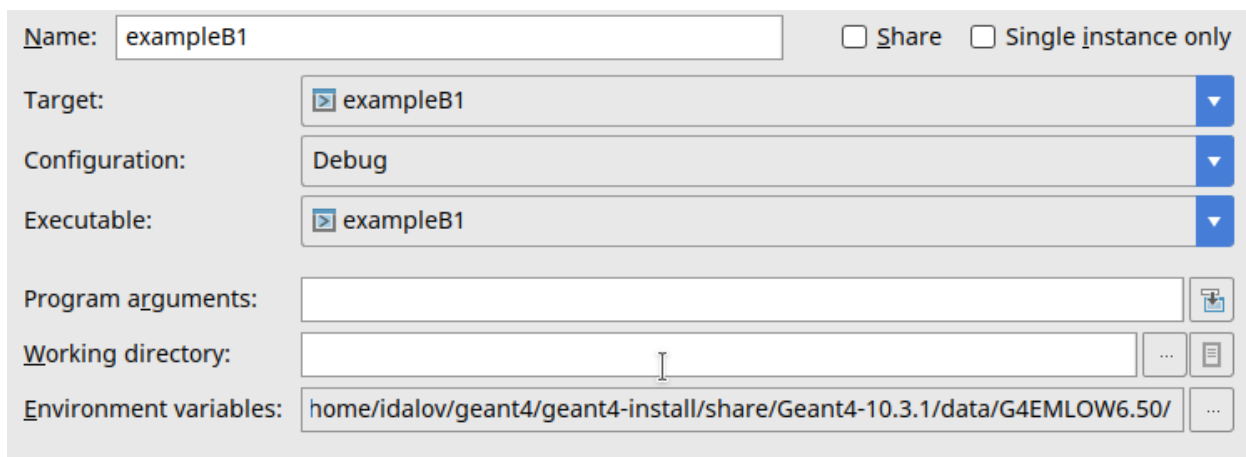
Примечание: В примере B1 для корректной работы необходимы следующие наборы данных с кодовыми названиями G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA и G4ENSDFSTATEDATA, но в дальнейшем рекомендуется включать все возможные библиотеки, так как зависимости для каждого из рассматриваемых примеров в данной книге указываться не будут.

Указать пути к необходимым данным в Clion можно следующим образом:

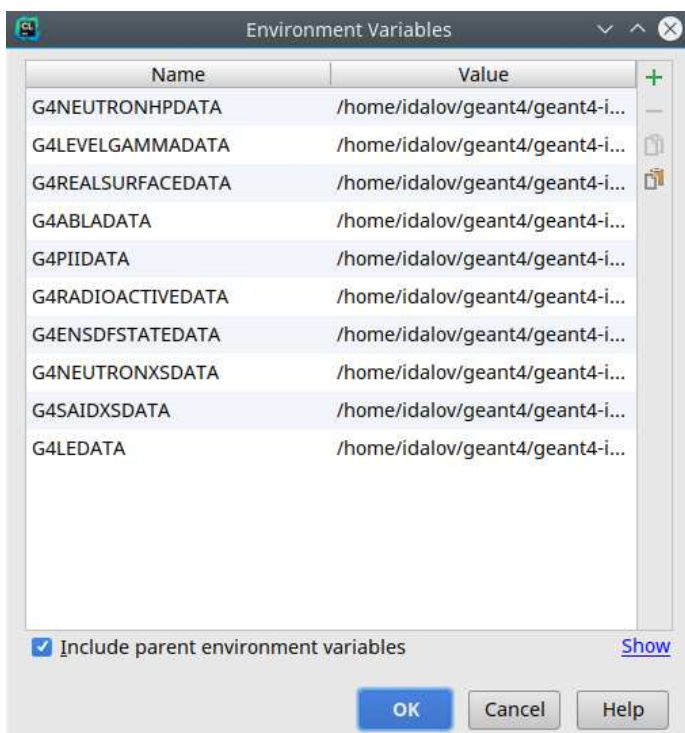
Шаг 3.1 Открываем меню Edit Configurations...



Шаг 3.2 В открытом окне указываем путь к необходимым данным, нажав на «...», как указано на рисунке:

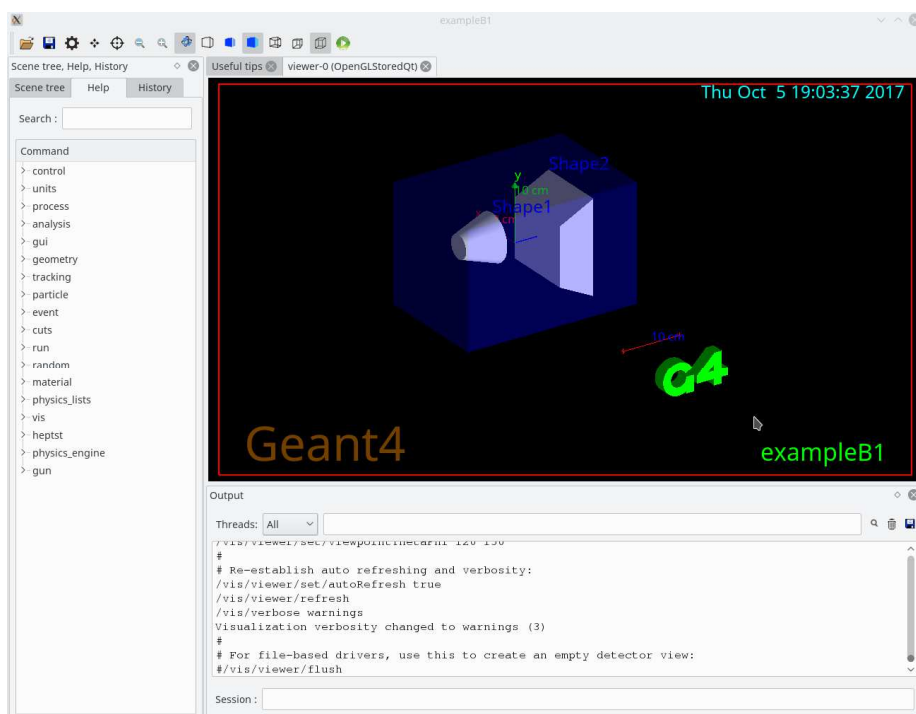


Шаг 3.3 Указываем необходимые пути.



Шаг 4. Запускаем пример

Теперь можно запустить пример. Если все было сконфигурировано верно, то в результате запустится основное рабочее окно Geant4:



2.RunManager

Под названием Geant4 скрываются сотни классов, решающих ту или иную задачу. Однако связь между собой имеет лишь ограниченное число, базовых, а часто абстрактных базовых классов, вместе образующих ядро Geant4. Для инициализации ядра в Geant4 существует специальный класс G4RunManager. Этот класс является управляющим. Он контролирует последовательность выполнения программы, а также управляет циклом событий во время запуска. Кроме того, пользователь может использовать иную версию данного класса — G4MTRunManager, если работает в многопоточном режиме.

Рассмотрим пример инициализации ядра Geant4. Для простоты следует включить в начало нашей программы следующий код:

```
#include <G4RunManager.hh>

int main() {
    G4RunManager runManager;
    return 0;
}
```

В CmakeLists стоит указать путь до Geant4 и связать его с проектом:

```
set(NAME test)
project(${NAME})

set(CMAKE_PREFIX_PATH /home/idalov/geant4/geant4-install)

find_package(Geant4 REQUIRED)
include(${Geant4_USE_FILE})

add_executable(${NAME} main.cpp)
```

Также не стоит забывать, что для корректного выполнения данного кода необходимо указать все пути до файлов данных. В противном случае на экране отобразится сообщение об ошибке

```
----- EEEE ----- G4Exception-START ----- EEEE -----
*** G4Exception : PART70000
      issued by : G4NuclideTable
G4ENSDFSTATEDATA environment variable must be set
```

В случае успешного выполнения программы сообщение будет выглядеть следующим образом:

```
*****
Geant4 version Name: geant4-10-04-patch-02 [MT] (25-May-2018)
      Copyright : Geant4 Collaboration
      References : NIM A 506 (2003), 250-303
                  : IEEE-TNS 53 (2006), 270-278
                  : NIM A 835 (2016), 186-225
                  WWW : http://geant4.org/
*****
```

В большинстве примеров Geant4 используется конструкция следующего вида:

```
#ifdef G4MULTITHREADED
    auto runManager = new G4MTRunManager;
#else
    auto runManager = new G4RunManager;
```

Как можно заметить, за счет значения G4MULTITHREADED выбирается, какой конструктор нужно использовать для создания runManager. В данном случае это «многопоточный» режим.

Перейдем к обзору основных методов данного класса. Как было сказано ранее, одной из функций runManager является связь пользовательских классов с ядром G4. С этой целью предоставлена группа перегрузок метода SetUserInitialization.

```
virtual void SetUserInitialization(G4VUserPhysicsList* userPL);
virtual void SetUserInitialization(G4VUserDetectorConstruction* userDC);
virtual void SetUserInitialization(G4VUserActionInitialization* userInit);
```

В G4 для корректной работы любого приложения необходимо передать информацию о трех основных столпах моделирования, а именно о том: «Где

происходит моделирование?», «По каким законам происходит моделирование?», «Как происходит моделирование?». На первый вопрос отвечает класс геометрии (подробнее в следующей главе). На второй — класс, содержащий список всех физических процессов, которые могут произойти в рамках данного моделирования, и наконец, на третий вопрос отвечает класс пользовательских действий, включающий информацию о том, какие первичные частицы порождаются.

Таким образом, за счет метода `SetUserInitialization()` нужно передать объекту класса `G4MTRunManager/G4RunManager` указатели на все интересующие объекты.

```
runManager->SetUserInitialization(new Geometry());           //геометрия
runManager->SetUserInitialization(new Shielding);           //процессы
runManager->SetUserInitialization(new Action());             //классы действий
```

После того как вся необходимая информация передана, следует вызвать метод инициализации (`Initialize()`).

```
virtual void Initialize();
```

Как было сказано, `RunManager` является управляющим классом. Это означает, что через него никак не связанные элементы моделирования могут получить доступ к другим частям проекта.

Получить доступ к уже существующему `runManager` можно за счет соответствующего статического Get-метода.

```
static G4RunManager* GetRunManager();
```

Можно воспользоваться любым из Get-методов данного класса, возвращающих указатели на соответствующие части моделирования

```
inline const G4VUserDetectorConstruction* GetUserDetectorConstruction() const
{ return userDetector; }
inline const G4VUserPhysicsList* GetUserPhysicsList() const
{ return physicsList; }
inline const G4VUserActionInitialization* GetUserActionInitialization() const
{ return userActionInitialization; }
```

Отдельного упоминания достоин метод `SetVerboseLevel`.

Данный метод отвечает за уровень подробности, выводимой менеджером информации. К примеру, по умолчанию уровень выводимой информации 0. Если с помощью данного метода установить значение 2, то появится возможность увидеть куда больше информации о том, что происходит внутри

```
inline void SetVerboseLevel(G4int vl)
```

Geant4.

Например, можно заметить, что необходимости удаления большинства создаваемых объектов (предусмотренных ядром G4, разумеется) нет. И все классы удаляются одновременно с разрушением объекта `G4RunManager`.

2. VisManager

В Geant4 для упрощения процесса отладки, а также с целью большей наглядности процесса моделирования представлен визуальный режим работы.

Если в рамках проекта необходимо визуальное представление, то следует создать объект класса `G4VisExecutive`, наследующий `G4VisManager`.

```
G4VisExecutive (const G4String& verbosityString = "warnings");
```

В зависимости от типа установки Geant4 могут быть доступны различные средства отображения, в рамках данного пособия используется комбинация QT4 и OpenGL. Остальные доступные средства представления, которые могут отличаться в зависимости от операционной системы, представлены в основном руководстве по Geant4, где указана их применимость и особенности.

Для инициализации графической оболочки недостаточно создать объект класса, а, как и в случае с `RunManager`, необходимо вызвать метод инициализации

```
void Initialize ();
```

Однако не стоит считать, что сразу после вызова метода `Initialize()`, появится сцена, содержащая в себе все элементы моделирования.

Стоит отметить, что `G4RunManager` и `G4VisManager` независимы, а, следовательно, их разрушение нужно производить отдельно.

2.UImanager

С целью повышения гибкости приложений, реализованных с использованием Geant4, был предусмотрен особый режим командной строки, а также группа команд.

Для того, чтобы получить возможность работы с командами, следует воспользоваться статическим методом, возвращающим указатель на объект соответствующего класса.

```
static G4UImanager * GetUImanager();
```

Подробнее использование пользовательских команд будет рассмотрено в соответствующем разделе, однако, стоит отметить, что для включения интерактивного режима необходимо создать объект класса `G4UIExecutive`. В качестве аргументов объект данного класса принимает стандартные аргументы главной функции `main(int argc, char**argv)`

```
G4UIExecutive(G4int argc, char** argv, const G4String& type = "");
```

Для запуска интерактивного режима работы следует воспользоваться методом

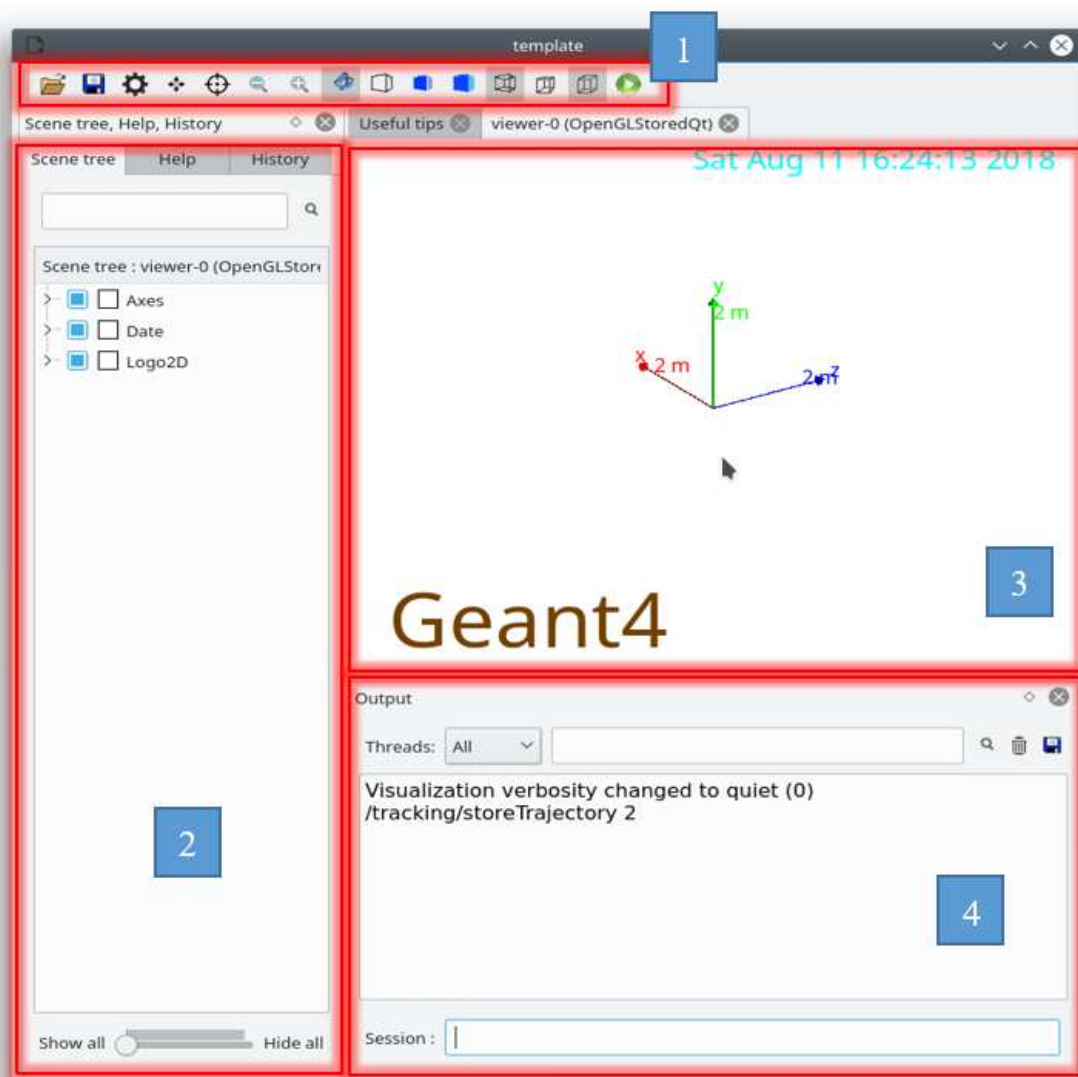
```
void SessionStart();
```

По завершению работы не стоит забывать удалять все созданные объекты классов управления:


```
delete ui;  
delete visManager;  
delete runManager;
```

2. Рабочее окно визуального интерактивного режима

В большинстве примеров, рассмотренных в рамках данного пособия, используется комбинация Qt4 и OpenGL для визуального представления программы моделирования. В этом случае рабочее окно при запуске программы будет выглядеть следующим образом (рисунок):



Под номером **1** находится панель, содержащая базовые кнопки по управлению программой на основе Geant4.

	открыть макрос -файл
---	----------------------

	сохранить текущую сцену в макрос файл
	посмотреть настройки сцены
	сместить сцену
	- открыть окно, чтобы узнать свойства выбранного объекта
	уменьшить масштаб
	увеличить масштаб
	поворачивать сцену
	оставить только ребра объектов, скрывая за невидимыми поверхностями
	показать ребра и поверхности объектов
	показывать все поверхности
	показывать все ребра
	показать сцену в перспективе
	показать сцену в ортогональной проекции
	запустить одно событие

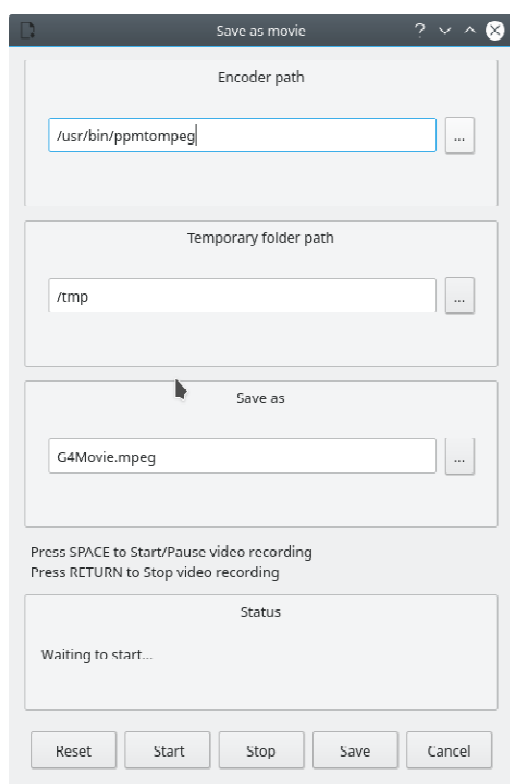
под номером 2 находятся вкладки Scene tree, Help и History

- вкладка Scene tree содержит все графические элементы, отраженные на сцене, включая логотипы, геометрические объекты, оси и т. п.
- вкладка Help содержит все доступные в интерактивном режиме работы команды, включая пользовательские
- вкладка History отражает все вызванные в рамках данной сессии команды

под номером **3** находится сцена, на которой отображается весь процесс моделирования

под номером **4** находится командная строка и история выводимых сообщений.

Если нажать на клавиатуре клавишу Enter, можно вызвать окно захвата видео со сцены.



2,5. Единицы измерения

К единицам измерения в Geant4 представлен особый подход. Для некоторых единиц измерения было установлено значение «1.». К ним относятся

- millimeter (mm) - миллиметр
- nanosecond (ns) - наносекунда
- MeV - Мега электронвольт
- eplus - заряд позитрона
- kelvin - кельвин
- mole - моль
- candela -сила света
- radian - радиан
- steradian - стерадиан

Всё остальное задаётся через указанные в этом списке величины. Все единицы измерения описаны в файле G4SystemOfUnits.hh.

Рассмотрим пример:

Если необходимо передать значения длины в см, то следует поступить следующим образом:

```
double length = 5. * cm;  
G4cout << length << '\n';
```

Выводимым в консоль результатом будет 50.

Рассмотрим еще один пример: грамм не является величиной по умолчанию в Geant4. Он представлен как

```
static constexpr double kilogram = joule*second*second/(meter*meter);  
static constexpr double gram = 1.e-3*kilogram;
```

где

```
static constexpr double joule = electronvolt/e_Sl;    // joule = 6.24150 e+12 * MeV
```

Откуда ожидаемо, что выводимое в следующем примере значение

```
double mass = 5. * g;  
G4cout << mass << '\n';
```

составит

3.12075e+22

Для вывода того или иного значения в единицах, необходимых для удобного визуального представления, следует разделить выводимое значение на необходимую величину т. е.:

```
double mass = 5. * g;  
G4cout << mass / g << '\n';
```

В данном случае результат будет 5, т. е. заданное изначально значение в граммах.

Также если существует необходимость в добавлении единиц измерения, то следует расширить заголовочный файл G4SystemOfUnits.hh. Например, можно добавить дюймы следующим образом:

```
#include <G4SystemOfUnits.hh>  
  
static const G4double inch = 2.54*cm;
```

Наконец, чтобы увидеть весь список доступных единиц измерения, можно воспользоваться командой:

```
/units/list
```

3. Создание материалов.

Структура материалов в Geant4 отражает то, что реально существует в природе: материалы состоят из отдельных элементов или их смесей, элементы в свою очередь состоят из отдельных изотопов или их смесей. Более подробно материалы в Geant4 рассмотрим на примере моделирования воды.

Как было сказано выше, создание материала начинается с моделирования изотопов, входящих в состав элементов, образующих целевой материал. За изотопы отвечает класс G4Isotope.

```
G4Isotope(const G4String& name,    //имя
          G4int z,                 //атомный номер
          G4int n,                 //количество нуклонов
          G4double a = 0.,         //молярная масса
          G4int m = 0);            //изомерный уровень
```

Как видно из конструктора класса, обязательными полями являются: имя, атомный номер и количество нуклонов. Молярная масса рассчитывается автоматически на этапе конструирования объекта (если не указано значение по умолчанию).

Соответственно, создание изотопа водорода(протий) в простейшем случае будет выглядеть следующим образом:

```
auto isoH = new G4Isotope("1H",1,1);
```

Теперь перейдем к созданию элемента "Водород". Конструктор элемента в Geant4 имеет следующий вид:

```
G4Element(const G4String& name,    //имя
          const G4String& symbol,   //символ
          G4int nblsotopes);        //количество изотопов
```

Необходимо отметить, что имя может быть любым, а символ должен соответствовать символу элемента в таблице Менделеева, к примеру:

```

auto protium = new G4Isotope("1H", 1, 1);
auto deuterium = new G4Isotope("2H", 1, 2);

auto elH_1 = new G4Element("my_hydrogen_1", "H", 1);
elH_1->AddIsotope(protium, 1.0);

auto elH_2 = new G4Element("my_hydrogen_2", "H", 1);
elH_2->AddIsotope(deuterium, 1.0);

```

Здесь создается 2 элемента, один из которых на 100% состоит из протия, а другой на 100% из дейтерия. В обоих случаях это водород, поэтому symbol у них "H". Кроме того, стоит обратить внимание на метод AddIsotope:

```

void AddIsotope(G4Isotope* isotope,           //указатель на изотоп
                G4double RelativeAbundance);  //массовая доля изотопа

```

Для каждого создаваемого элемента данный метод должен вызываться ровно столько раз, сколько изотопов было указано в конструкторе.

Существует альтернативный конструктор элементов.

При использовании этого конструктора не нужно создавать изотопы напрямую.

```

G4Element(const G4String& name,           //имя
          const G4String& symbol,        //символ
          G4double Zeff,                 //атомный номер
          G4double Aeff);                //молярная масса

```

Они будут сконструированы автоматически, исходя из природного соотношения, хранимого в базе.

Теперь перейдем к созданию материалов. За материалы отвечает класс G4Material. Конструктор, позволяющий создавать материалы из созданных ранее элементов выглядит следующим образом:

```

G4Material(const G4String& name,           //имя
           G4double density,               //плотность
           G4int nComponents,              //количество компонентов
           G4State state = kStateUndefined, //состояние
           G4double temp = NTP_Temperature, //температура
           G4double pressure = CLHEP::STP_Pressure); //давление

```

Вначале рассмотрим поля *state*, *temp* и *pressure*. Поле *state* отвечает за "состояние" материала. Оно может быть: твердым, жидким, газообразным или "неопределенным".

Данное поле принимает значение в виде эnumерации, где

0	<i>kStateUndefined</i>
1	<i>kStateSolid</i>
2	<i>kStateLiquid</i>
3	<i>kStateGas</i>

По умолчанию значения температуры(*temp*) и давления(*pressure*) соответствуют нормальным условиям.

Рассмотрим теперь поле *nComponents*. Оно отвечает за количество компонентов в материале, причем под компонентами могут пониматься как отдельные элементы, так и уже готовые материалы.

Теперь можно перейти непосредственно к моделированию воды. Код будет выглядеть следующим образом:

```
auto Water = new G4Material("Water", 1 * g / cm3, 2);
Water->AddElement(elH, 2);
Water->AddElement(elO, 1);
```

Метод `AddElement` имеет две перегрузки. В данном случае используется

```
void AddElement(G4Element* element, G4int nAtoms);
```

версия,

принимающая целочисленное количество атомов в молекуле.

Кроме того, у данного метода есть иная форма:

```
void AddElement (G4Element* element, G4double fraction);
```

В данном случае в качестве второго аргумента передается массовая доля элемента в конечном материале. Соответственно, при использовании данной перегрузки исходный код по созданию воды примет следующую форму:

```
auto Water2 = new G4Material("Water2", 1 * g / cm3, 2);
Water2->AddElement(elH, 11.19*perCent);
Water2->AddElement(elO, 88.81*perCent);
```

Аналогично элементам для материалов предусмотрен конструктор, который позволяет пропустить стадию создания изотопов и элементов. Это достаточно легкий способ генерации "моно-материалов" (материалов, состоящих из одного элемента).

```
auto mat_O = new G4Material("Oxygen", 8., 16. * g / mole, 1. * g / cm3);
```

```
auto mat_H = new G4Material("Hydro_", 1., 1. * g / mole, 1. * g / cm3);
```

В Geant4 материалы можно создавать не только из элементов, но и из уже созданных ранее других материалов. К примеру, из созданных выше моно-материалов водорода и кислорода можно получить воду следующим образом:

```
auto Water3 = new G4Material("Water3", 1 * g / cm3, 2);  
Water3->AddMaterial(mat_H, 11.19 * perCent);  
Water3->AddMaterial(mat_O, 88.81 * perCent);
```

Где метод `AddMaterial` аналогичен перегрузке метода `AddElement`, принимающей в качестве аргумента массовую долю.

Наконец, для получения необходимого количества компонентов материала можно комбинировать вызовы методов `AddMaterial` и `AddElement`:

```
auto Water4 = new G4Material("Water4", 1 * g / cm3, 2);  
Water4->AddMaterial(mat_H, 11.19 * perCent);  
Water4->AddElement(elO, 88.81 * perCent);
```

Однако стоит отметить, что использование перегрузки метода `AddElement` с количеством атомов, и любых методов с массовым содержанием, может привести к некорректному результату.

В завершении стоит сказать, что при создании материала из других материалов не учитываются промежуточные свойства его компонентов. В конечном итоге, материал состоит из элементов, а его плотность, температура и т.п. определяются только финальным объектом.

Рассмотрим еще один конструктор класса `G4Material`

```
G4Material(const G4String& name,           //имя
           G4double density,               //плотность
           const G4Material* baseMaterial, //базовый материал
           G4State state = kStateUndefined, //состояние
           G4double temp = NTP_Temperature, //температура
           G4double pressure = CLHEP::STP_Pressure); //давление
```

Данный конструктор позволяет создавать материалы с новыми свойствами (плотность, состояние, температура, давление) из уже существующих.

```
auto new_Water = new G4Material("new_Water", 5 * g / cm3, Water, kStateLiquid, 1000 * kelvin);
```

3. База материалов NIST.

В Geant4 представлена база материалов, составленная NIST Physical Measurement Laboratory. В данной базе доступно более 3000 изотопов, а все представленные элементы составлены исходя из природного баланса изотопов. Элементы доступны от Водорода до Калифорния (98). Кроме того, в базе доступны различные готовые материалы, такие как:

- составные вещества и смеси (ткань, эквивалентная пластику; морской воздух и т.д.)
- биохимические материалы (жировая ткань, цитозин, тимин и т.д.)
- композитные материалы (например, кевлар)

Для того, чтобы получить доступ к базе NIST, следует воспользоваться статическим методом

```
static G4NistManager* Instance();
```

Данный метод возвращает указатель, с помощью которого можно обращаться к шаблонам уже готовых материалов, например, водород можно получить следующим образом

```
auto nist = G4NistManager::Instance();
auto nist_H_1 = nist->FindOrBuildElement("H");
```

В данном случае метод FindOrBuildElement позволяет найти элемент в базе по его символу, однако существует перегрузка для доступа по атомному номеру:

```
auto nist_H_2 = nist->FindOrBuildElement(1);
```

Для поиска готовых материалов в качестве ключа следует использовать имя материала. Полный список доступных материалов в Geant4 можно найти в разделе: Geant4 Book For Application Developers > Appendix > Geant4 Material Database

Например, получаемый таким образом моно-материал "кислород"

```
auto nist_O_mat = nist->FindOrBuildMaterial("G4_O");
```

создаваемую в предыдущей части воду можно получить как

```
auto nist_Water = nist->FindOrBuildMaterial("G4_WATER");
```

В завершение следует сказать, что создаваемые вручную изотопы, элементы и материалы должны иметь уникальные имена в рамках своей группы. Однако при повторных вызовах nist-методов FindOrBuild*(Material/Element) возвращается указатель на материал/элемент, который был создан при первом вызове, и ошибок не возникает.

4. Геометрия.

В основе геометрии Geant4 лежит концепция логических и физических объемов. Логический объем представляет собой элемент детектора определенной формы заданного материала, чувствительности и т. п. Кроме того, каждый логический объем может содержать внутри другие объемы. Физический объем представляет собой пространственное расположение логического объема относительно логического материнского объема (Подробнее ниже).

Для реализации геометрии в Geant4 необходимо унаследовать абстрактный базовый класс `G4VUserDetectorConstruction`. Если открыть описание шаблона данного класса, то можно заметить, что он содержит один

```
virtual G4VPhysicalVolume* Construct() = 0;
```

чисто виртуальный, т. е. обязательный для реализации, метод.

Чтобы понять, на что возвращается указатель в данном методе, рассмотрим, как реализуется геометрия в Geant4.

4. Формы

Любой геометрический объект строится в 3 этапа:

- На первом этапе создается основа будущего логического объема - его форма, включающая геометрические характеристики.
- На втором - описываются свойства формы, такие как: материал, сцинтилляционные свойства и т. п. Кроме того, на втором этапе форме можно установить различные визуальные атрибуты, упрощающие дальнейшую визуализацию моделирования.
- На третьем этапе осуществляется расположение модифицированной формы в пространстве, её поворот, смещение и т. п.

Начнем с рассмотрения примитива - прямоугольного параллелепипеда, или, проще говоря, коробки. За реализацию данной формы отвечает класс G4Box.

Чтобы создать простейшую форму – коробку, следует воспользоваться конструктором:

```
G4Box(const G4String& pName, G4double pX, G4double pY, G4double pZ); // Конструктор  
коробки с именем, и половинами длин по pX,pY,pZ
```

Сразу отметим, что в качестве значений по X, Y, Z принимаются ПОЛОВИНЫ от реальной длины, ширины и высоты. Данное свойство характерно для большинства форм, используемых в G4.

Соответственно, реализация объекта простейшей формы будет выглядеть следующим образом:

```
G4Box *box = new G4Box("box", 5*cm, 5*cm, 5*cm);
```

Все геометрические формы в Geant4 реализованы в соответствии с концепцией Constructive Solid Geometry (CSG). Большинство комплексных форм описывается за счет их граничной поверхности, которая может быть первого, второго порядка, или B-spline поверхностью. Все это сделано, опираясь на ISO STEP стандарт CAD систем.

Возвращаясь к базовым формам, в Geant4 представлен целый набор примитивных форм, являющихся потомками абстрактного класса G4SCGSolid. К этому списку можно отнести: коробки, сферы, конусы, трубки и др.

Кроме того, для всех базовых форм предусмотрены методы для расчета занимаемого геометрического объема созданной ранее коробки:

```
G4cout << "Volume = " << box->GetCubicVolume() << '\n';
```

а также её площади поверхности:

```
G4cout << "Surface Area = " << box->GetSurfaceArea() << '\n';
```

4. Логические объемы

Сама по себе форма не содержит никакой информации об объекте, кроме его геометрических размеров. Однако для моделирования взаимодействия излучения с веществом необходимо описать свойства этого самого вещества, в котором осуществляется моделирование. С этой целью в Geant4 предоставлен класс объектов, называемый логический объем. Одной из основных целей логического объема является связь формы с физическими свойствами объекта, как пример, материал, из которого она(форма) сделана.

Объекты логического объема относятся к классу `G4LogicalVolume`. Конструктор данного класса выглядит следующим образом:

```
G4LogicalVolume(G4VSolid* pSolid,           //форма
                G4Material* pMaterial,       //материал
                const G4String& name,         //имя
                G4FieldManager* pFieldMgr=0,
                G4VSensitiveDetector* pSDetector=0,
                G4UserLimits* pULimits=0,
                G4bool optimise=true);
```

Стоит отметить, что поля «форма и материал» (`pSolid` and `pMaterial`) не должны быть равными `nullptr`. Поля: `pFieldMgr`, `pSDetector`, `pULimits` являются опциональными и в данной работе описаны не будут.

Рассматривая формы, следует обратить внимание на визуализацию геометрических объектов, например цвет или прозрачность, так как визуальные атрибуты являются частью именно логической, а не физической формы (см. ниже).

Для того, чтобы изменить цвет того или иного геометрического объекта, следует вызвать метод:

```
void SetVisAttributes (const G4VisAttributes& VA);
```

который в качестве аргумента принимает интересующий нас параметр визуализации.

Рассмотрим простейший куб с длиной грани 10 см.

```
G4double box_size = 5 * cm;  
G4Box *box = new G4Box("box", box_size, box_size, box_size);  
  
auto box_log = new G4LogicalVolume(box, nist->FindOrBuildMaterial("G4_SODIUM_IODIDE"),  
                                     "box_LOG");
```

```
box_log->SetVisAttributes(G4Colour::Blue());
```

В качестве формы используется «коробка». Объект log_box представляет собой логический объем, в качестве аргументов - принимающий форму и моделируемый материал. Кроме того, в рассматриваемом примере осуществляется передача логическому объему визуального атрибута «синий цвет». Стоит отметить, что теперь все физические объемы, которые будут использовать в качестве параметра данный логический объем, окрасятся синим цветом.

Отдельно стоит отметить имена форм и логических объемов. В рамках объектов своего класса эти имена должны быть уникальны. Для формы, логического объема и физического объема одного геометрического объекта можно использовать одно имя.

Как и для формы, для логического объема предусмотрено несколько полезных методов. Один из них позволяет рассчитать массу:

```
G4cout << "Mass = " << box_log->GetMass() / g << " g\n";
```

Для получения результата, к примеру в граммах, следует разделить получаемый за счет метода GetMass() результат на «грамм».

Mass = 3667 g

4. Физический объем. Материнский объем

Последним этапом построения геометрического объекта средствами Geant4 является реализация его физического объема или, иначе говоря, расположение в пространстве.

Geant4 предусматривает несколько способов осуществления данного процесса. Рассмотрим самый простейший среди них. В качестве инструмента создания физического объема воспользуемся конструктором G4PVPlacement (наследует G4VPhysicalVolume)

```
G4PVPlacement(G4RotationMatrix *pRot,           //матрица поворота
               const G4ThreeVector &tlate,       //вектор смещения
               G4LogicalVolume *pCurrentLogical, //логический объем
               const G4String& pName,            //имя
               G4LogicalVolume *pMotherLogical,  //материнский объем
               G4bool pMany,                     //не используется
               G4int pCopyNo,                   //номер копии
               G4bool pSurfChk=false);          //проверка на
                                                //пересечение с другими объемами
```

где pRot — матрица поворота, tlate — вектор смещения, pCurrentLogical — логический объем, pName — имя (идентификатор), pMotherLogical — материнский объем, pMany — данный параметр не реализован в текущей версии Geant4 (можно использовать false), pCopyNo - номер копии логического объема, для первого физического объема следует использовать значение 0, затем 1 и т.д.

Рассмотренный конструктор позволяет расположить в пространстве моделирования единичный логический объем, а также повернуть и сместить его относительно материнского объема.

Все геометрические объекты в Geant4 позиционируются внутри созданных ранее других объектов. Соответственно каждый создаваемый

физический объем смещается относительно центра логического объема, называемого материнским для данного физического объема.

Не трудно догадаться, что должен существовать один единственный материнский логический объем, который становится изначальным или главным для всех последующих объемов.

```
world_PV = new G4PVPlacement(nullptr, G4ThreeVector(), world_log, "world_PV", nullptr, false, 0);
```

Также такой объем называют миром.

В качестве указателя на материнский объем для этого физического объема следует использовать нулевой указатель.

Помним, что единственный необходимый для реализации геометрии метод:

```
virtual G4VPhysicalVolume* Construct() = 0;
```

возвращает указатель на физический объем. Объем, на который возвращается указатель, является главным материнским объемом или представленным выше миром, с привязанными к нему всеми дочерними объемами, используемыми в геометрии.

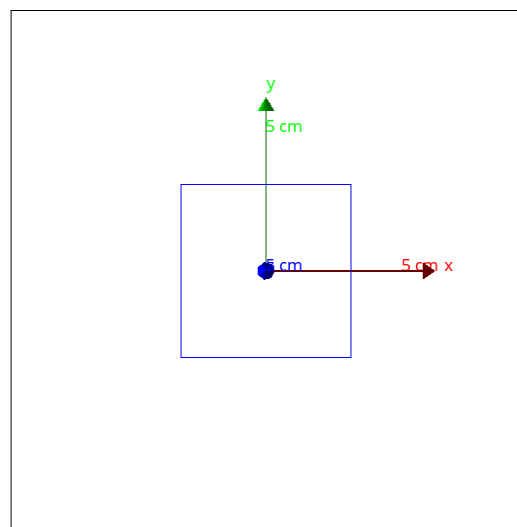
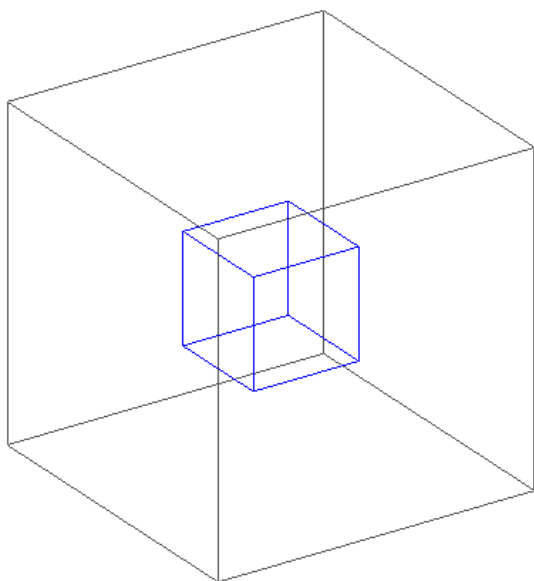
Рассмотрим пример реализации метода Construct() для размещения представленной выше коробки в мире.

```
G4VPhysicalVolume* Geometry::Construct() {
    auto size = 15 * cm;
    auto world = new G4Box("world", size / 2., size / 2., size / 2.);
    auto world_log = new G4LogicalVolume(world, world_mat, "world_log");
    world_log->SetVisAttributes(G4VisAttributes::Invisible);
    auto world_PV = new G4PVPlacement(nullptr, G4ThreeVector(), world_log, "world_PV",
                                     nullptr, false, 0);

    G4double box_size = 2.5 * cm;
    auto *box = new G4Box("box", box_size, box_size, box_size);
    auto box_log = new G4LogicalVolume(box, nist->FindOrBuildMaterial("G4_SODIUM_IODIDE"),
                                     "box_LOG");
    box_log->SetVisAttributes(G4Colour::Blue());
    new G4PVPlacement(nullptr, G4ThreeVector(0, 0, 0), box_log, "box_PV", world_log, false, 0);
    return world_PV;
}
```

В данном примере мир представляет собой коробку 15x15x15 см. С нулевым смещением и без поворота внутри данного мира размещается реализованная

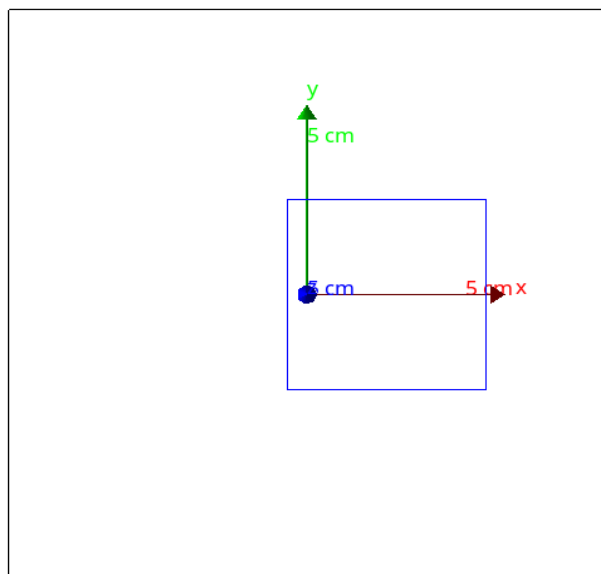
выше синяя коробка 5x5x5 см. Если визуализировать данную геометрию средствами G4, будет получен следующий результат:



Попробуем сместить коробку относительно центра мира на 2 см по оси X. Для этого в качестве аргументов вектора смещения следует передать значения (2*cm,0,0)

```
new G4PVPlacement(nullptr, G4ThreeVector(2 * cm, 0, 0), box_log, "box_PV", world_log, false, 0);
```

В результате:



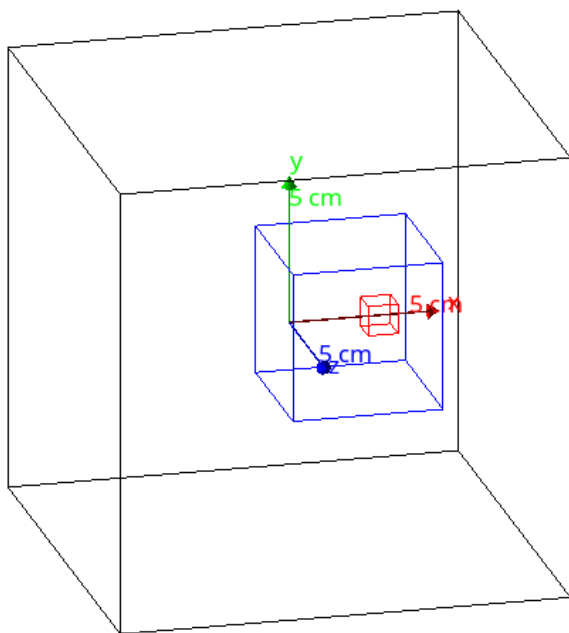
```

auto box_1 = new G4Box("box_1", box_size_1, box_size_1, box_size_1);
auto box_1_log = new G4LogicalVolume(box_1, nist->FindOrBuildMaterial("G4_BGO"), "box_1");
box_log->SetVisAttributes(G4Colour::Red());
new G4PVPlacement(nullptr, G4ThreeVector(1 * cm, 0, 0), box_1_log, "box_1", box_log, false, 0);

```

Усложним задачу и добавим внутри синего куба еще один, дочерний для него.

Соответственно, в качестве материнского объема теперь выступает не **world_log** а **box_log**. Кроме того, осуществим смещение нового кубика относительно материнского на 1*см по оси X. В результате:

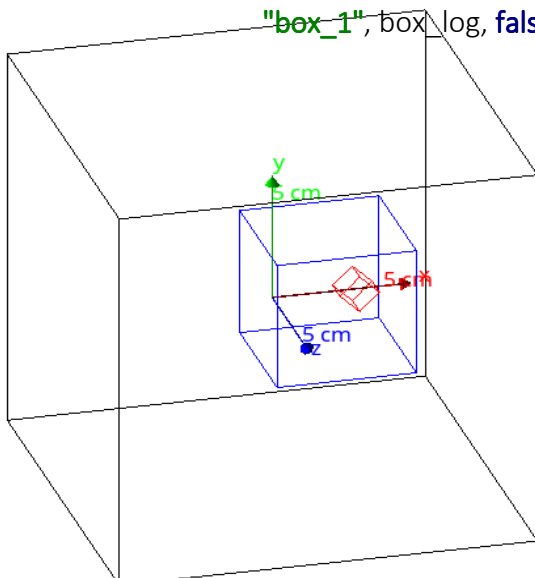


Можно заметить, что смещение красного кубика произошло не в глобальной системе координат мира, а в системе координат синего кубика. Для большей наглядности осуществим поворот сначала красного кубика на 45 градусов за счет угла phi,

```

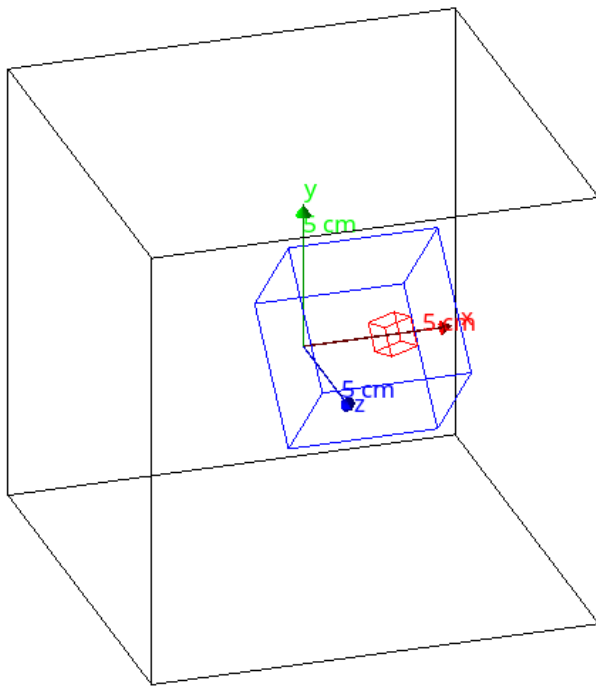
new G4PVPlacement(new G4RotationMatrix(45 * deg, 0, 0), G4ThreeVector(1 * cm, 0, 0), box_1_log,
"box_1", box_log, false, 0);

```



а затем синего на 45 градусов с помощью угла teta

```
new G4PVPlacement(new G4RotationMatrix(0, 45 * deg, 0), G4ThreeVector(2 * cm, 0, 0), box_log,  
                  "box_PV", world_log, false, 0);
```



Как можно видеть, поворот осуществляется за счет передачи указателя на объект класса матрицы поворота, где
первое поле — phi,
второе поле — teta
третье поле — psi

4. Контейнеры-хранилища

Все создаваемые в процессе моделирования формы, логические объемы и физические объемы помещаются в специальные контейнеры-хранилища для удобного доступа к элементам геометрии из других частей проекта.

- Для форм — G4SolidStore
- Для логических объемов — G4LogicalStore
- Для физических объемов — G4PhysicalVolumeStore.

Рассмотрим содержимое хранилища. Для получения доступа к контейнеру необходимо вызвать статический метод `GetInstance()`, возвращающий указатель на выбранное хранилище. Так как по своей природе хранилище является контейнером, то с ним можно работать с помощью итераторов. Например, распечатаем имена всех созданных в проекте форм:

```
for (auto item : *G4SolidStore::GetInstance())
    G4cout << item->GetName() << '\n';
```

```
world
box
box_1
```

в результате в консоль выведется следующая информация:

Аналогичную информацию можно получить для хранилищ логических и физических объемов.

Если мы добавим объем с уже существующим именем-идентификатором, то он успешно добавится в общий список хранилища. К примеру, добавив дополнительный физический объем, используя уже существующий «красный кубик» с тем же именем, что было задано ранее:

```
new G4PVPlacement(new G4RotationMatrix(45 * deg, 0, 0), G4ThreeVector(1 * cm, -3 * cm, 0),
    box_1_log, "box_1", box_log, false, 1);
```

и распечатав содержимое хранилища физических форм:

```
for (auto item : *G4PhysicalVolumeStore::GetInstance())
    G4cout << item->GetName() << '\n';
```

в консоли будет выведено:

```
world_PV
box_PV
box_1
box_1
```

Однако, данная ситуация может вызвать проблемы в некоторых задачах моделирования. Например, если возникнет необходимость идентификации конкретного детектирующего элемента в массиве.

При добавлении нового физического объема из уже существующего ранее логического объема, дублирования данного логического объема не возникает:

```
for (auto item : *G4LogicalVolumeStore::GetInstance())  
    G4cout << item->GetName() << "\n";
```

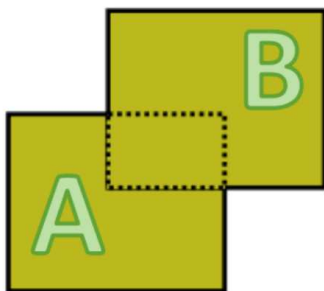
В КОНСОЛИ:

```
world_log  
box_LOG  
box_1
```

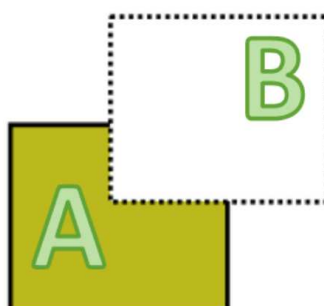
4. Логические операции над формами

Кроме инструментов для создания простейших геометрических форм в Geant4 предоставлен набор логических операций, позволяющих реализовывать сложные пространственные объекты.

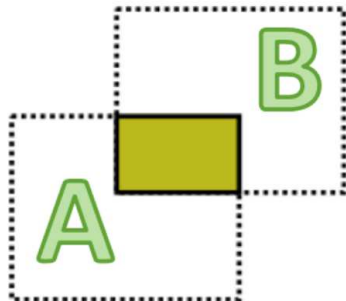
Данные операции принято называть логическими. К ним относятся



G4UnionSolid — объединение двух геометрических форм A и B в одну;



G4SubtractionSolid — вычитание из одной формы A другой формы B;



G4IntersectionSolid — Общая область пересечения формы A и формы B.

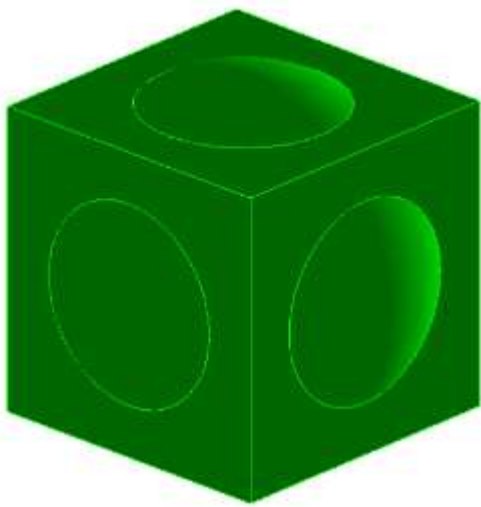
Все логические формы наследуют класс G4BooleanSolid, который, в свою очередь, наследует класс G4VSolid. Отсюда следует, что любую логическую форму можно использовать как обычную форму для построения логического объема.

Рассмотрим примеры построения геометрических объемов, формы которых созданы за счет логических операций.

Допустим дан куб (G4Box) с длинной грани 10*см и шар (G4Orb) радиуса 6*см. Тогда их пересечение можно описать в виде:

```
G4double box_len = 10 * cm;  
G4double orb_r = 6 * cm;  
  
;auto box = new G4Box("box", box_len / 2., box_len / 2., box_len / 2.);  
auto orb = new G4Orb("orb", orb_r);  
  
auto my_union = new G4UnionSolid("union", box, orb);  
auto my_union_log = new G4LogicalVolume(my_union, nist->FindOrBuildMaterial("G4_BGO"),  
                                         "union");  
  
my_union_log->SetVisAttributes(G4Color::Green());  
  
new G4PVPlacement(nullptr, G4ThreeVector(), my_union_log, "union", world_log, false, 0);
```

Полученная конструкция будет иметь вид, представленный на рисунке



Аналогично, для тех же начальных данных G4SubtractionSolid можно использовать следующим образом:

```
auto box = new G4Box("box", box_len / 2., box_len / 2., box_len / 2.);
auto orb = new G4Orb("orb", orb_r);

auto my_subtraction = new G4SubtractionSolid("subtraction", box, orb);
auto my_subtraction_log = new G4LogicalVolume(my_subtraction, nist->FindOrBuildMaterial("G4_BGO"),
                                              "subtraction");

my_subtraction_log->SetVisAttributes(G4Color::Green());

new G4PVPlacement(nullptr, G4ThreeVector(), my_subtraction_log, "subtraction", world_log, false, 0);
```

с результатом, представленным на рисунке.



И наконец, для операции пересечения

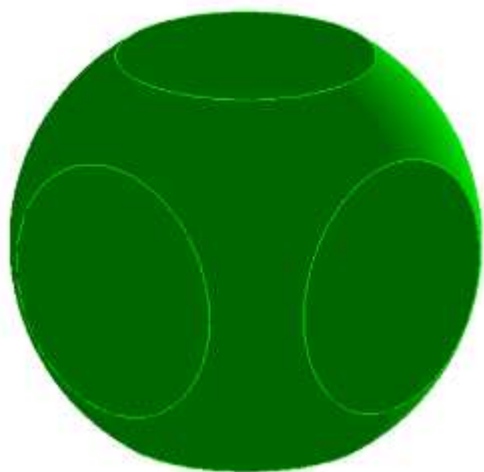
```
auto box = new G4Box("box", box_len / 2., box_len / 2., box_len / 2.);
auto orb = new G4Orb("orb", orb_r);

auto my_intersection = new G4IntersectionSolid("intersection", box, orb);
auto my_intersection_log = new G4LogicalVolume(my_intersection, nist->FindOrBuildMaterial("G4_BGO"),
                                                "intersection");

my_intersection_log->SetVisAttributes(G4Color::Green());

new G4PVPlacement(nullptr, G4ThreeVector(), my_intersection_log, "intersection", world_log, false, 0);
```

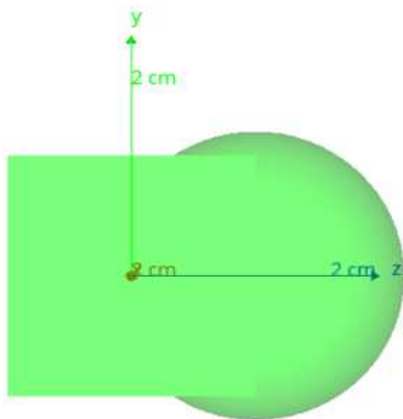
в результате получится следующее изображение



При создании логических форм можно использовать как смещение, так и поворот. Всегда следует иметь ввиду, что все операции осуществляются относительно центра первой в методе формы.

```
auto my_union = new G4UnionSolid("union", box, orb, nullptr, G4ThreeVector(0, 0, 10. * cm));
```

Центром полученной таким образом логической фигуры всегда является центр первой фигуры. Результат полученного выше объединения, расположенного в начале координат, можно увидеть на рисунке.



4. Контейнеры для логических объемов

Представим, что в рамках задачи в геометрии многократно используется один и тот же сложный массив объектов. С целью упрощения построения таких геометрий в Geant4 реализованы специальные контейнеры. Они позволяют группировать логические объемы с целью многократного воспроизведения в мире.

Класс, отвечающий за данные контейнеры, называется `G4AssemblyVolume`. Данный класс имеет два конструктора:

```
G4AssemblyVolume();
G4AssemblyVolume( G4LogicalVolume* volume,      //логический объем
                  G4ThreeVector& translation,    //смещение объема
                  G4RotationMatrix* rotation);    //поворот объем
```

В случае применения параметризованного конструктора смещение и поворот осуществляются относительно центра контейнера.

Существует метод на добавление логического объема в уже существующий контейнер.

```
void AddPlacedVolume( G4LogicalVolume* pPlacedVolume, //размещаемый объем
                    G4ThreeVector& translation,        //смещение
                    G4RotationMatrix* rotation);      //поворот
```

В контейнер можно добавить другой, ранее созданный контейнер.

Для размещения контейнера в материнском мире следует воспользоваться

```
void AddPlacedAssembly( G4AssemblyVolume* pAssembly,      //размещаемый контейнер
                        G4ThreeVector& translation,        //смещение
                        G4RotationMatrix* rotation);       //поворот

void MakeImprint( G4LogicalVolume* pMotherLV,             //материнский объем
                  G4ThreeVector& translationInMother,     //смещение в мат. объеме
                  G4RotationMatrix* pRotationInMother,   //поворот в мат. объеме
                  G4int copyNumBase = 0,                  //номер копии
                  G4bool surfCheck = false );             //проверка на пересечение
```

методом:

При многократном добавлении одного и того же логического объема в контейнер новые логические копии этого объема не появляются. Однако, все размещенные таким образом физические объемы будут иметь уникальные имена

Рассмотрим пример, в котором один и тот же куб трижды добавляется в контейнер. В каждой итерации осуществляется смещение куба по одной из осей на длину его грани.

Код данной задачи будет выглядеть следующим образом:

```
G4double box_len = 10 * cm;
auto box = new G4Box("box", box_len / 2., box_len / 2., box_len / 2.);
auto box_log = new G4LogicalVolume(box, nist->FindOrBuildMaterial("G4_BGO"),
                                   "box");

box_log->SetVisAttributes(G4Color::Green());

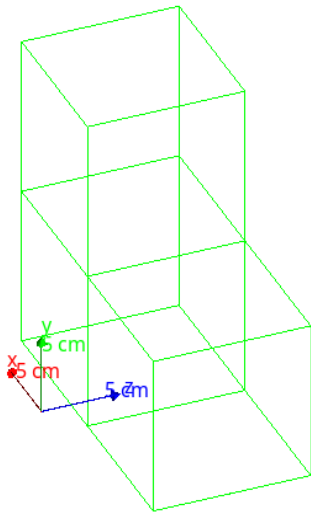
G4ThreeVector pos_av_vect(0, 0, 10 * cm);

auto AV = new G4AssemblyVolume(box_log, pos_av_vect, nullptr);

pos_av_vect.setX(10 * cm);
AV->AddPlacedVolume(box_log, pos_av_vect, nullptr);
pos_av_vect.setY(10 * cm);
AV->AddPlacedVolume(box_log, pos_av_vect, nullptr);

pos_av_vect.set(0, 0, 0);
AV->MakeImprint(world_log, pos_av_vect, nullptr);
```


Итоговая конфигурация будет соответствовать рисунку.



Если вывести имена физических объемов, доступные в данной задаче, то в консоли будет получен следующий результат.

```
world_PV
av_1_impr_1_box_pv_0
av_1_impr_1_box_pv_1
av_1_impr_1_box_pv_2
```

Также стоит отметить, что если расположить данный контейнер в мире еще один раз, финальный список физических объемов примет следующую форму:

```
world_PV
av_1_impr_1_box_pv_0
av_1_impr_1_box_pv_1
av_1_impr_1_box_pv_2
av_1_impr_2_box_pv_0
av_1_impr_2_box_pv_1
av_1_impr_2_box_pv_2
```

5. Физические процессы

Второй обязательной частью реализации математической модели посредством Geant4 является построение списка используемых физических процессов. Данный аспект модели отвечает на вопрос: «По каким законам осуществляется моделирование?».

В Geant4 представлен огромный набор первичных частиц и процессов взаимодействия между ними и веществом. В основе реализации лежит класс `G4VUserPhysicsList`, однако сразу стоит отметить его расширение `G4VModularPhysicsList`. В случае расширения все используемые процессы группируются в модели, а из них, в свою очередь, формируются списки физических процессов.

Вне зависимости от выбранного способа конечный объект класса-списка должен быть передан ядру за счет метода:

```
virtual void SetUserInitialization(G4VUserPhysicsList* userPL);
```

5. Готовые списки физических процессов

Начать рассмотрение данного раздела было бы уместно с укомплектованных разработчиками Geant4 списков физических процессов. Дело в том, что задачи кафедры «Прикладная ядерная физика» редко выходят за границу областей интересов:

- нейтронов (от тепловых, до быстрых с энергией порядка 20 МэВ),
- а также альфа, бета и гамма источников.

Потребности большинства задач с лихвой покрываются следующими моделями:

QBBC — включает полный набор моделей для альфа-, бета-, гамма-излучений, содержит процессы связанные с упругим/неупругим взаимодействием, тормозным излучением и многое другое. Однако данная модель не применима для нейтронного взаимодействия.

*_HP — любые модели, содержащие в конце своего названия данное сокращение. К примеру, QGSP_BERT_HP или QGSP_BIC_HP. Сокращение HP в данном случае означает, что данные списки содержат высокоточные модели для нейтронов низких энергий. Различия же в таких моделях чаще всего

начинаются в высокоэнергетических диапазонах, и выходят за рамки задач кафедры.

Полное описание содержимого данных моделей во много раз превысит объем данного пособия, поэтому приведено не будет. Однако с ним можно ознакомиться на официальном сайте в разделах D и F (Physics Reference Manual и Physics List Guide) соответственно.

Для использования данных списков необходимо и достаточно передать объекты их классов ядру Geant4:

```
auto runManager = new G4MTRunManager;  
  
runManager->SetUserInitialization(new Geometry());  
runManager->SetUserInitialization(new QBBC);           // использование QBBC  
runManager->SetUserInitialization(new Action());
```

Так же стоит упомянуть, что в основе нейтронной модели лежат данные по сечениям, базирующиеся на ENDF/B-VII.r1.

5. Сборка физического списка

Иногда возникают задачи, для решения которых стандартных списков физических процессов оказывается недостаточно. Рассмотрим основные аспекты построения пользовательского списка физических процессов. В качестве базового класса удобнее будет использовать G4VModularPhysicsList, особенно, если имеется необходимость переписывать не весь список, а только его часть.

Данный класс содержит метод

```
void RegisterPhysics(G4VPhysicsConstructor* );
```

в качестве аргумента у которого используется модель физических процессов.

Оригинальный QBBC, основанный на G4VModularPhysicsList выглядит, к примеру, следующим образом:

```

QBBC::QBBC( G4int ver, const G4String& )
{
  G4DataQuestionaire it(photon, neutronxs);
  G4cout << "<<< Reference Physics List QBBC " << G4endl;

  defaultCutValue = 0.7*mm;
  SetVerboseLevel(ver);

  // EM Physics
  RegisterPhysics( new G4EmStandardPhysics(ver) );

  // Synchrotron Radiation & GN Physics
  RegisterPhysics( new G4EmExtraPhysics(ver) );

  // Decays
  RegisterPhysics( new G4DecayPhysics(ver) );

  // Hadron Physics
  RegisterPhysics( new G4HadronElasticPhysicsXS(ver) );

  RegisterPhysics( new G4StoppingPhysics(ver) );

  RegisterPhysics( new G4IonPhysics(ver) );

  RegisterPhysics( new G4HadronInelasticQBBC(ver));

```

где G4int - параметр ver отвечает за «уровень» подробности выводимой в консоль информации. Уровень 0 — минимальный, 1 (реже 2) — максимальный.

Перейдем к рассмотрению класса — модели G4VPhysicsConstructor. Данный класс содержит два чисто виртуальных метода:

```
virtual void ConstructParticle()=0;
```

```
virtual void ConstructProcess()=0;
```

ConstructParticle() отвечает за первичную инициализацию частиц. Следовательно, абсолютно все частицы, используемые в процессах, объявленных в данной модели, должны быть инициализированы в этом методе. Для инициализации частиц возможно два пути. Для стандартных частиц (гамма, электрон, нейтрон, протон и т. п.) можно воспользоваться статическими методами:

```
G4Gamma::GammaDefinition();  
G4Neutron::NeutronDefinition();  
G4Electron::ElectronDefinition();  
G4Proton::ProtonDefinition();
```

Однако, в случае наличия таких частиц как ионы, барионы и т. п. используются статические методы `ConstructParticle()`, позволяющие сконструировать все частицы своей группы:

```
G4IonConstructor::ConstructParticle();  
G4MesonConstructor::ConstructParticle();  
G4ShortLivedConstructor::ConstructParticle();
```

Посредством `ConstructProcess()` осуществляется подключение всех используемых процессов к модели. В данном случае удобнее будет рассмотреть его реализацию на примере.

Пример использования метода `ConstructProcess()`

Реализуем подключение High Precision Models (HP) для нейтронов низких энергий (до 20 МэВ), а именно, подключение процессов упругого и неупругого взаимодействия нейтронов.

За упругое взаимодействие отвечает

```
G4HadronElasticProcess(const G4String& procName = "hadElastic");
```

За неупругое

```
G4NeutronInelasticProcess(const G4String& processName = "neutronInelastic");
```

Сразу стоит отметить, что рассматриваемые выше процессы наследуют класс `G4VDiscreteProcess`, произошедший в свою очередь от `G4VProcess` — базового класса всех процессов Geant4.

Первое, что нужно сделать в случае реализации выше рассматриваемых процессов, это указать используемые наборы данных сечений (наследуют `G4VCrossSectionDataSet`).

Будем использовать два различных набора сечений: один для диапазона 4эВ — 20МэВ, второй для тепловых нейтронов вплоть до энергии 4эВ:

```
auto theNeutronElasticProcess = new G4HadronElasticProcess;
```

```
theNeutronElasticProcess->AddDataSet(new G4ParticleHPElasticData());
```

```
theNeutronElasticProcess->AddDataSet(new G4ParticleHPThermalScatteringData);
```

Далее следует указать специальные модели для процесса (т. к. оригинальный процесс рассчитан на весь диапазон энергий (т. е. до ТэВ))

Для этого предусмотрен метод:

```
void RegisterMe(G4HadronicInteraction* a);
```

где в качестве аргумента принимается специальная модель (только для адронов)

```
auto theNeutronElasticModel = new G4ParticleHPElastic;
```

```
theNeutronElasticModel->SetMinEnergy(4.0*eV);
```

```
theNeutronElasticProcess->RegisterMe(theNeutronElasticModel);
```

```
theNeutronThermalElasticModel = new G4ParticleHPThermalScattering;
```

```
theNeutronThermalElasticModel->SetMaxEnergy(4.0*eV);
```

```
theNeutronElasticProcess->RegisterMe(theNeutronThermalElasticModel);
```

в данном случае методы:

```
inline void SetMinEnergy( G4double anEnergy );
```

```
inline void SetMaxEnergy( const G4double anEnergy );
```

отвечают за диапазон действия той или иной модели в процессе.

После того как настройки процесса закончены, его нужно передать соответствующему типу частиц. В данном примере это будет выглядеть как:

```
G4ProcessManager* pmanager = G4Neutron::Neutron()->GetProcessManager();
```

```
pmanager->AddDiscreteProcess(theNeutronElasticProcess);
```

Объект класса `G4ProcessManager` является классом - контейнером, содержащим все используемые в программе процессы. У каждого типа частиц существует свой контейнер. Для подключения дискретного процесса в демонстрируемом примере используется метод:

```
G4int G4ProcessManager::AddDiscreteProcess(G4VProcess *aProcess);
```

Что касается неупругого взаимодействия, то ограничимся одной моделью на весь исследуемый диапазон энергии, т. е.:

```
auto theNeutronInelasticProcess = new G4NeutronInelasticProcess();

theNeutronInelasticProcess->AddDataSet(new G4ParticleHPInelasticData());

auto theNeutronInelasticModel = new G4ParticleHPInelastic;
theNeutronInelasticProcess->RegisterMe(theNeutronInelasticModel);

pmanager->AddDiscreteProcess(theNeutronInelasticProcess);
```

Особенности физических списков в Geant4.10+

Geant4.10 и выше позволяет работать в многопоточном режиме. Это особенность несколько изменит код, представленный в примере. Объекты классов-процессов и их компоненты объединяются в специальную структуру, являющуюся статической и одновременно локальной для каждого рабочего потока.

Сама структура будет выглядеть следующим образом:

```
struct ThreadPrivate {
    G4HadronElasticProcess* theNeutronElasticProcess;
    G4ParticleHPElastic* theNeutronElasticModel;
    G4ParticleHPThermalScattering* theNeutronThermalElasticModel;
    G4NeutronInelasticProcess* theNeutronInelasticProcess;
    G4ParticleHPInelastic* theNeutronInelasticModel;
};
```

а статическое поле

```
static G4ThreadLocal ThreadPrivate* tpdata;
```

Сразу стоит отметить, что так как поле статическое, инициализировать его надо аналогично глобальной переменной, т. е. вне пределов функций и классов:

```
G4ThreadLocal Test_HP::ThreadPrivate* Test_HP::tpdata = 0;
```

Что же касается самого класса, то он примет вид

```

class Test_HP : public G4VPhysicsConstructor{
private:
    struct ThreadPrivate {
        G4HadronElasticProcess* theNeutronElasticProcess;
        G4ParticleHPElastic* theNeutronElasticModel;
        G4ParticleHPThermalScattering* theNeutronThermalElasticModel;
        G4NeutronInelasticProcess* theNeutronInelasticProcess;
        G4ParticleHPInelastic* theNeutronInelasticModel;
    };
    static G4ThreadLocal ThreadPrivate* tpdata;

public:
    void ConstructParticle() override {
        G4Gamma::GammaDefinition();
        G4Neutron::NeutronDefinition();
        G4Electron::ElectronDefinition();
        G4Proton::ProtonDefinition();
        G4IonConstructor::ConstructParticle();
        G4MesonConstructor::ConstructParticle();
        G4ShortLivedConstructor::ConstructParticle();
    }

    void ConstructProcess() override {
        if ( tpdata == 0 ) tpdata = new ThreadPrivate;
        tpdata->theNeutronElasticProcess = new G4HadronElasticProcess;

        tpdata->theNeutronElasticProcess->AddDataSet(new G4ParticleHPElasticData());
        tpdata->theNeutronElasticProcess->AddDataSet(new G4ParticleHPThermalScatteringData());
        tpdata->theNeutronElasticModel = new G4ParticleHPElastic;
        tpdata->theNeutronElasticModel->SetMinEnergy(4.0*eV);
        tpdata->theNeutronElasticProcess->RegisterMe(tpdata->theNeutronElasticModel);

        tpdata->theNeutronThermalElasticModel = new G4ParticleHPThermalScattering;
        tpdata->theNeutronThermalElasticModel->SetMaxEnergy(4.0*eV);
        tpdata->theNeutronElasticProcess->RegisterMe(tpdata->theNeutronThermalElasticModel);

        tpdata->theNeutronInelasticProcess = new G4NeutronInelasticProcess();
        tpdata->theNeutronInelasticProcess->AddDataSet(new G4ParticleHPInelasticData());
        tpdata->theNeutronInelasticModel = new G4ParticleHPInelastic;
        tpdata->theNeutronInelasticProcess->RegisterMe(tpdata->theNeutronInelasticModel);

        G4ProcessManager* pmanager = G4Neutron::Neutron()->GetProcessManager();

        pmanager->AddDiscreteProcess(tpdata->theNeutronElasticProcess);
        pmanager->AddDiscreteProcess(tpdata->theNeutronInelasticProcess);
    }
};

```


Стоит отметить, что методы `ConstructParticle()` и `ConstructProcess()` доступны для `G4VUserPhysicsList` и `G4VModularPhysicsList` соответственно. Однако в случае этих классов чисто виртуальными они не являются.

6. Классы действий

Как было сказано ранее, в Geant4 существуют два основных класса инициализации свойств моделирования: первый отвечает на вопрос ГДЕ осуществляется моделирование(геометрия), а второй - КАК и по каким законам это моделирование происходит (физические процессы). Кроме того, существует группа классов «Действий» над частицей в процессе моделирования. Главным среди них является класс, определяющий КАКИМ ОБРАЗОМ, генерируются частицы, а его реализация становится обязательным шагом при написании программы для моделирования взаимодействия частиц с веществом на основе Geant4. В добавлении к нему существует группа необязательных классов действий, позволяющих извлекать пользователю информацию на различных этапах моделирования, анализировать её и аккумулировать.

Инициализация всех классов действий отличается от способа, используемого для геометрии и физических процессов. Дело в том, что, начиная с версии Geant4.10, была реализована поддержка многопоточности. С этого момента геометрия и физические процессы остались общей информацией, существующей в единственном экземпляре на мастер-потоке, а объекты классов действий стали уникальными для каждого потока - рабочего.

В связи с этим в Geant4 появился класс G4VUserActionInitialization, основным назначением которого является создание объектов классов действий для рабочих потоков.

Рассмотрим основные классы действий и способ их инициализации.

К классам действий относятся:

- G4VUserPrimaryGeneratorAction
- G4UserRunAction
- G4UserEventAction
- G4UserStackingAction
- G4UserTrackingAction
- G4UserSteppingAction

Их инициализация осуществляется за счет **protected** методов `SetUserAction()`.

Данные методы вызываются в методе `Build()` потомка `G4VUserActionInitialization` для создания объектов рабочих потоков.

Например, потомок классов `G4VUserActionInitialization` может выглядеть

```
class my_Actions : public G4VUserActionInitialization {
public:
    void Build() const override {
        SetUserAction(new my_run_action);
        SetUserAction(new my_stack_action);
        SetUserAction(new my_event_action);
        SetUserAction(new my_track_action);
        SetUserAction(new my_step_action);

        SetUserAction(new my_primary_gen);
    }
};
```

следующим образом:

где `my_*` - потомки классов действий, объекты которых инициализируются для рабочих потоков.

Если же необходимо (допустимо только для `UserRunAction`) создать объект только для мастер-потока, то следует вызвать метод `SetUserAction()` в методе `BuildForMaster()` вместо метода `Build()`, однако в рамках данной книги этот способ рассмотрен не будет.

6. Генерация первичных событий

Обязательным для реализации в программе моделирования является только один класс действий — `G4VUserPrimaryGeneratorAction`.

Данный абстрактный класс содержит один чисто виртуальный метод:

```
virtual void GeneratePrimaries(G4Event* anEvent) = 0;
```

Однако, чтобы понять принцип работы данного класса, рассмотрим некоторые понятия, связанные с генерацией первичных частиц в Geant4.

Первичная частица и её трек, а также все вторичные частицы и их треки, образовавшиеся в процессе взаимодействия первичной частицы с веществом называются событием.

Первичное положение частицы в пространстве математической модели называется вершиной.

Класс `G4VUserPrimaryGeneratorAction` отвечает за связь между создаваемыми пользователем первичными частицами и их запуском в рамках модели. Сам по себе данный класс никаких первичных частиц не создает. Для этой цели в Geant4 служит базовый класс `G4VPrimaryGenerator` и группа его потомков. Для простоты опустим архитектуру класса `G4VPrimaryGenerator` и сразу перейдем непосредственно к рассмотрению одного из его потомков `G4ParticleGun` — самый простой и универсальный генератор частиц.

`G4ParticleGun` позволяет одновременно настраивать как первичную частицу, так и её вершину. Для это в данного классе представлены следующие методы:

для первичной частицы:

1. Установка типа первичной частицы

```
void SetParticleDefinition (G4ParticleDefinition * aParticleDefinition);
```

2. Кинетическая энергия первичной частицы

```
void SetParticleEnergy(G4double aKineticEnergy);
```

```
void SetParticleMomentum(G4double aMomentum);
```

3. Импульс частицы

либо

```
void SetParticleMomentum(G4ParticleMomentum aMomentum);
```

где в качестве аргумента `G4ParticleMomentum` может использоваться объект класса `G4ThreeVector`.

4. Направление импульса

```
void SetParticleMomentumDirection(G4ParticleMomentum aMomentumDirection);
```

5. Заряд частицы (если доступно)

```
void SetParticleCharge(G4double aCharge);
```

6. Поляризация (если доступно)

```
void SetParticlePolarization(G4ThreeVector aVal);
```

Для вершины первичной частицы представлены следующие методы:

1. Позиция вершины

```
void SetParticlePosition(G4ThreeVector aPosition);
```

2. Стартовое время

```
void SetParticleTime(G4double aTime);
```

Кроме всего прочего, класс `G4ParticleGun` имеет три варианта конструкторов:

```
G4ParticleGun();  
G4ParticleGun(G4int numberofparticles);  
G4ParticleGun(G4ParticleDefinition * particleDef, G4int numberofparticles = 1);
```

где `numberofparticles` отвечает за количество частиц, запускаемых из первичной вершины с текущими начальными свойствами в рамках одного события.

Теперь более подробно рассмотрим методы 2,3,4 для первичной частицы.

Если направление и энергия первичной частицы не заданы с помощью методов, то ей будет присвоено направление вдоль оси X (1,0,0) с энергией в 1000 МэВ.

Метод `SetParticleMomentum(G4ParticleMomentum aMomentum)` в качестве аргумента принимает `G4ThreeVector` (ибо `G4ParticleMomentum` есть typedef от `G4ThreeVector`). Направление вектора будет соответствовать направлению частицы, а его длина — энергии из расчета, что единица длины это 1 МэВ.

Однако для лучшей читаемости кода рекомендуется использовать пару методов `SetParticleEnergy(G4double)` и `SetParticleMomentumDirection(G4ParticleMomentum)`, где первый отвечает за кинетическую энергию частицы, а второй только за её направление без влияния на энергию.

В качестве примера рассмотрим моноэнергетический точечный изотропный гамма - источник.

Настройки `G4ParticleGun` будут выглядеть следующим образом:

```
gun = new G4ParticleGun(1);
gun->SetParticleDefinition(G4Gamma::GammaDefinition());
gun->SetParticlePosition(G4ThreeVector(0, 0, 0));
gun->SetParticleEnergy(661 * keV);
gun->SetParticleMomentumDirection(G4ThreeVector(G4UniformRand(), G4UniformRand(),
G4UniformRand()));
```

где `G4UniformRand()` возвращает случайное дробное число от 0 до 1.

Теперь рассмотрим, как передать заданные настройки для моделирования. Метод `GeneratePrimaries(G4Event*anEvent)` класса `G4VUserPrimaryGenerationAction` вызывается единожды на каждом отдельно взятом событии. Для того, чтобы передать настройки моделирования первичной

частицы, следует вызвать метод `GeneratePrimaryVertex(G4Event*)` для используемого генератора.

Соответственно, возвращаясь к выше указанным настройкам, метод `GeneratePrimaryVertex(G4Event*)` примет следующую форму.

```
void PrimaryPat::GeneratePrimaries(G4Event* anEvent) {  
    auto gun = new G4ParticleGun(1);  
    gun->SetParticleDefinition(G4Gamma::GammaDefinition());  
    gun->SetParticlePosition(G4ThreeVector(0, 0, 0));  
    gun->SetParticleEnergy(661 * keV);  
    gun->SetParticleMomentumDirection(G4ThreeVector(G4UniformRand(), G4UniformRand(),  
G4UniformRand()));  
  
    gun->GeneratePrimaryVertex(anEvent);  
}
```

Стоит отметить что, если в качестве аргумента конструктора `G4ParticleGun` указать значение больше, чем 1, все запускаемые в рамках одного события частицы будут иметь одинаковые свойства (включая направление, а новые случайные значения будут разыгрываться только для частиц, запускаемых на следующем событии).

6. Пример использования энергетического спектра для генератора первичного излучения.

Как было показано выше, на каждом отдельном событии генератор имеет вполне конкретное значение кинетической энергии частицы. Посмотрим, как можно решить большинство типовых задач, в которых энергия задана функциональной зависимостью.

Пусть энергия частицы(e) на отрезке $[A, B]$ задана плотностью распределения $f(x)$, где $f(x)$ описана функцией

```
G4double foo(G4double e);
```

создадим контейнер типа `std::vector<G4double>` для формирования ключей конечной функции распределения.

```
keys = new std::vector<G4double>();
```

Выбрав шаг *de* необходимой точности, перейдем к дискретному распределению следующим образом:

записываем соответствующие значения функции в контейнер `keys` в выбранных точках,

в переменной `sum` накапливаем интеграл, для последующего перехода к

```
G4double buf, sum{0}, i;  
for (i = a; i < b; i += de) {  
    buf = foo(i);  
    sum += buf;  
    keys->push_back(buf);  
}
```

вероятностям

и преобразуем контейнер, переходя к вероятностям

Таким образом, мы получили контейнер с вероятностями для дискретной величины. Осуществим интегрирование в этом же цикле для перехода к функции распределения вероятности:

```
for (double &key : *keys)  
    key /= sum;  
*keys->begin()/=sum;  
for (auto item = ++keys->begin(); item!=keys->end();item++)  
    ((*item /= sum) += *(item-1));
```


и построим контейнер типа `std::map<G4double,G4double>`, который будет содержать в качестве ключей значения функции распределения, а в качестве значений выбранные ранее значения энергии. В результате получим

```
keys = new std::vector<G4double>();
result = new std::map<G4double, G4double>();

G4double buf, sum{0}, i;
for (i = a; i < b; i += de) {
    buf = foo(i);
    sum += buf;
    keys->push_back(sum);
}

*keys->begin()/=sum;
for (auto item = ++keys->begin(); item!=keys->end();item++)
    ((*item /= sum) += *(item-1));

auto item = keys->begin();
for (i = a; i < b; i += de)
    result->emplace(*item++, i);
```

Можно заметить, что если сначала осуществить интегрирование, а уже потом нормировку, код заметно упростится:

```
keys = new std::vector<G4double>();
result = new std::map<G4double, G4double>();

G4double buf, sum{0}, i;
for (i = a; i < b; i += de) {
    buf = foo(i);
    sum += buf;
    keys->push_back(sum);
}

auto item = keys->begin();
for (i = a; i < b; i += de)
    result->emplace(*item++ / sum, i);
```

Теперь для установки значения энергии по заданному закону достаточно воспользоваться методом

```
void SetParticleEnergy(G4double aKineticEnergy);
```

в следующей форме:

```
gun->SetParticleEnergy(result->upper_bound(G4UniformRand())->second);
```

7. Цикл обработки событий

В предыдущей главе было указано, что кроме одного обязательного для реализации класса действий существует группа опциональных классов действий. Данные классы позволяют пользователю управлять процессом анализа и аккумуляции информации, которая сопутствует процессу моделирования.

Чтобы разобрать назначение тех или иных классов, рассмотрим процесс моделирования в Geant4. Моделирование в Geant4 начинается с Запуска (Run). Свойства геометрии и физических процессов фиксируются ядром Geant4 и становятся неизменными. Пользователь указывает, сколько событий он хочет рассмотреть и запускает программу на «расчет». Запуск начинается с момента старта первого события, а заканчивается моделированием всех вторичных частиц последнего. После завершения запуска пользователь может вновь менять по своему усмотрению геометрию и свойства используемых физических процессов.

Как говорилось ранее, запуск состоит из событий. Понятие события рассматривалось в предыдущей главе. Напомним, события состоят из треков, и моделирование события заканчивается с моделированием последнего трека вторичной частицы в данном событии. Треки моделируются по одному, независимо и последовательно.

Последовательность моделирования треков задается с помощью реализации стека треков. Сам по себе трек содержит все свойства частицы в процессе моделирования прохождения сквозь вещество. Каждый трек знает имя частицы, хранит информацию о её времени жизни и изменении кинетической энергии и т. п.

Все треки состоят из шагов. Шаг — это мельчайшая единица моделирования. Он характеризует расстояние (время), за которое произошло изменение состояния частицы. Это может быть переход из одного объема в

другой, потери энергии на ионизацию, вылет за исследуемую область моделирования и т. п.

Пользователь напрямую не управляет ни одной из этих условных единиц моделирования, однако ему предоставлены классы действий, связанные со своей соответствующей единицей. В рамках данных классов пользователь может анализировать информацию в процессе моделирования, а подчас и вносить некоторые корректировки в процесс моделирования.

7. Шаги

Шаг является мельчайшей единицей моделирования и самой информативной единицей анализа. Для взаимодействия с информацией на шагах следует унаследовать класс `G4UserSteppingAction`. У данного класса есть несколько виртуальных методов, однако особого внимания достоин один из них,

```
virtual void UserSteppingAction(const G4Step*);
```

а именно:

Этот метод вызывается в конце каждого шага. В качестве аргумента данный метод принимает константный указатель на текущий шаг. Рассмотрим информацию, которую можно извлечь из этого шага.

```
G4StepPoint* GetPreStepPoint() const;  
G4StepPoint* GetPostStepPoint() const;
```

Это два самых универсальных метода, содержащих информацию о состоянии частицы в начале и конце шага соответственно. С помощью этих методов, а также информации, содержащейся в объекте `G4StepPoint`, можно узнать такие вещи, как позиция, объем, в котором находилась/оказалась частица, её энергия и т. п.

К примеру, можно проверить, изменился ли в этом шаге объем:

```
if (aStep->GetPostStepPoint()->GetPhysicalVolume() != nullptr)  
    if (aStep->GetPostStepPoint()->GetPhysicalVolume() !=  
        aStep->GetPreStepPoint()->GetPhysicalVolume())  
        G4cout << "True\n";
```

Примечание: Стоит обратить внимание на проверку существования объема. Дело в том, что в случае вылета частицы за пределы области моделирования, объема в финальной точке не существует, что может привести к ошибке времени выполнения.

Существует группа методов, позволяющая определить изменения во времени, позиции и энергии:

```
G4double GetDeltaTime() const;  
G4ThreeVector GetDeltaPosition() const;  
G4double GetTotalEnergyDeposit() const;
```

К следующим полезным методами класса G4Step стоит отнести:

```
G4int GetNumberOfSecondariesInCurrentStep() const;  
const G4TrackVector* GetSecondary() const ;
```

Первый из методов возвращает количество вторичных треков, образовавшихся на текущем шаге. Второй метод дает указатель на `std::vector<G4Track*>()` скрытый под `typedef G4TrackVector`, следовательно с данным указателем можно работать как с обычным указателем на вектор.

Также не трудно предположить, что раз через шаг можно получить доступ к трекам вторичных частиц, то можно получить доступ к треку самой рассматриваемой частицы. Для этого предназначен метод:

```
G4Track* GetTrack() const;
```

7. Треки

Трек содержит максимально полную информацию. Он содержит информацию о типе частицы, процессе, в результате которого она образовалась и ID родителя. За счет трека можно определить, в каком объеме находится

частица. В треке содержится значение кинетической энергии, а также время существования частицы от момента создания и т. п.

Для получения этой информации следует воспользоваться следующими методами для определения:

кинетической энергии:

```
G4double G4Track::GetKineticEnergy() const;
```

типа частицы (Также из этого объекта можно узнать имя частицы):

```
G4ParticleDefinition* GetDefinition() const;  
const G4String& GetParticleName() const;    //для определения имени из объекта  
                                           //G4ParticleDefinition
```

времени от начала события или от создания трека:

```
G4double GetGlobalTime() const;            // Время от начала события  
  
G4double GetLocalTime() const;            // Время от начала трека
```

Также можно получить информацию о модели и процессе, в следствие которых была создана частица

```
const G4String& GetCreatorModelName() const;    // имя модели  
G4int GetCreatorModelID() const;               //id модели  
G4int GetParentID() const;                     //id родителя  
G4VProcess* GetCreatorProcess() const;         //указатель на процесс-создатель
```

Можно получить информацию о поляризации, длине трека и т. д. и т. п.

В зависимости от поставленной задачи информацию из трека удобно получать одним из двух способов:

1. Как было показано выше, через шаг.
2. За счет соответствующего класса действий G4UserTrackAction

Класс G4UserTrackAction содержит два метода для перезагрузки:

```
virtual void PreUserTrackingAction(const G4Track*){};  
virtual void PostUserTrackingAction(const G4Track*){};
```

где метод `PreUserTrackingAction` вызывается в начале каждого трека (даже для первой первичной частицы в событии он вызывается уже после начала события), а метод `PostUserTrackingAction` в конце события.

7. Стек при работе с треками

Еще один класс действий - `G4UserStackingAction`, является чисто техническим классом для определения порядка обработки треков и т.п. В рамках данной книги данный класс рассмотрен не будет.

7. События

Неоднократно подчеркивалось, что событие представляет собой объединение треков первичных частиц, образованных в рамках однократного вызова метода `void G4VUserPrimarygeneratorAction::GeneratePrimaries(G4Event*)` и появляющихся в следствие их моделирования треков вторичных частиц.

Классом действий, связанным с `G4Event`, является `G4UserEventAction`. Аналогично классу действий над треками класс `G4userEventAction` имеет два метода для перегрузки

```
virtual void BeginOfEventAction(const G4Event* anEvent);  
virtual void EndOfEventAction(const G4Event* anEvent);
```

Метод `BeginOfEventAction` вызывается в начале каждого события, а метод `EndOfEventAction` в конце.

G4Event в отличие от G4Step и G4Track не хранит особой информации о процессе моделирования. К полезным стоит отнести методы получения информации о первичной вершине:

```
G4int GetNumberOfPrimaryVertex() const;           //количество вершин
G4PrimaryVertex* GetPrimaryVertex(G4int i=0) const; //доступ к вершине по номеру
```

Может оказаться полезным метод

```
G4int GetEventID() const;
```

который возвращает ID текущего события. К примеру, его можно использовать, чтобы выводить сообщение о начале i-го события с целью оценки оставшегося времени выполнения

```
G4int printable = 10;
if(anEvent->GetEventID()%printable==0)
    G4cout<<anEvent->GetEventID()<<G4endl;
```

(каждое 10-ое событие на экран будет выводиться его номер)

В остальном роль класса G4UserEventAction в программе определяется конечным пользователем и его конкретной программой.

7. Запуски

Самой крупной единицей моделирования является запуск. С началом запуска начинается цикл моделирования, а конец запуска совпадает с завершением цикла.

За запуски отвечает класс G4Run, а соответствующим классом действий является G4UserRunAction. В данном классе доступны два метода для перезагрузки

```
virtual void BeginOfRunAction(const G4Run* aRun);
virtual void EndOfRunAction(const G4Run* aRun);
```


вызываемых в начале и конце каждого запуска.

Также в данном классе доступен метод

```
virtual G4Run* GenerateRun();
```

позволяющий генерировать вместо стандартных объектов G4Run объекты его потомков.

Чтоже касается остальных функций данного класса, то, как и в случае с G4UserEventAction, они зависят от реализации конкретной программы.

7. Сбор информации с шагов в запуске

До сих пор были рассмотрены только личные особенности конкретных классов действий, однако не говорилось о взаимодействии между ними. Дело в том, что универсальной цепочки связи между ними в базовых классах нет. Все базовые классы действий содержат лишь конструкторы по умолчанию. Однако за счет особенностей наследования пользователь волен сам определить, какой класс и как будет связан с другими.

Ранее был рассмотрен класс G4VUserActionInitialization. В его методе Build() осуществляется генерация всех необходимых объектов для конкретного потока. Изменив конструкторы классов действий таким образом, что один из них будет принимать указатель на другой, можно получить необходимую связь между ними.

Допустим, мы хотим узнать, сколько протонов образуется в результате моделирования:

Нас не интересует, на каком треке и в рамках какого события образовался протон, а нужен лишь факт образования. Следовательно, мы будем на шагах проверять факт образования вторичной частицы с именем «proton» и передавать об этом информацию напрямую в запуск. Отсюда легко заметить, что нам необходимо осуществить связь между G4UserSteppingAction и G4UserRunAction. Для этого стоит передать указатель на потомка G4UserRunAction в объект потомка G4UserSteppingAction с помощью

```
#include "G4UserSteppingAction.hh"
#include "RunAction.hh"
```

```
class SteppingAction : public G4UserSteppingAction
{
public:
    explicit SteppingAction(RunAction*);

    void UserSteppingAction(const G4Step*) override;
private:
    RunAction* runAction;
};
```

конструктора:

и осуществить связь от потомка G4UserSteppingAction к потомку G4UserRunAction в методе Build()

```
void Action::Build()const {
    SetUserAction(new PrimaryPat);           //генератор первичных частиц

    auto runAction = new RunAction;
    SetUserAction(new SteppingAction(runAction));
    SetUserAction(runAction);
}
```

Теперь для решения поставленной задачи достаточно создать в RunAction целочисленную переменную - счетчик и метод по инкрементации её значения

```

class RunAction : public G4UserRunAction {
public:
    void EndOfRunAction(const G4Run *aRun) override;

    void BeginOfRunAction(const G4Run *aRun) override;

private:
    G4int counter;
public:
    void AddEvent() { counter++; }
};

```

В методе BeginOfRunAction значение счетчика будет обнуляться:

```

void RunAction::BeginOfRunAction(__attribute__((unused)) const G4Run *aRun) {
    counter = 0;
}

```

а в методе EndOfRunAction выводиться в консоль

```

void RunAction::EndOfRunAction(__attribute__((unused)) const G4Run *aRun) {
    G4cout << "Result = " << counter << '\n';
}

```

Наконец, само событие по образованию протонов будет отслеживаться в методе

```

void SteppingAction::UserSteppingAction(const G4Step * aStep) {
    if (aStep->GetNumberOfSecondariesInCurrentStep() != 0)
        for (auto &item: *aStep->GetSecondary())
            if (item->GetDefinition()->GetParticleName() == "proton")
                runAction->AddEvent();
}

```

В зависимости от особенностей той или иной задачи условия и связи между классами будут абсолютно разными. Универсального решения всех задач не существует.

Примечание: не следует забывать, что в случае расчета в многопоточном режиме финальный результат, выводимый в EndOfRunAction, соответствует тем результатам, которые были получены на конкретном рабочем потоке. Для получения общего результата по всем потокам данные нужно «сшивать».

8. Многопоточные приложения в Geant4

В Geant4, начиная с десятой версии, была реализована поддержка многопоточных приложений в рамках ОС UNIX.

Современные операционные системы уже достаточно долго поддерживают многозадачность, причем каждая из этих задач вполне может выполняться в несколько потоков. Однако такой подход к программированию существенно усложняет разработку. Не будем основательно углубляться в особенности многопоточного программирования и сконцентрируемся лишь на тех аспектах, которые встречаются в Geant4. До десятой версии в Geant4 все происходило последовательно: сначала инициировалось ядро, потом устанавливались настройки модели, такие как геометрия, процессы. Затем инициализировались действия, описывались первичные частицы и присваивались все опциональные объекты. По завершению настройки программа запускалась на расчет и каждое событие моделировалось тоже последовательно.

Отличие работы в многопоточном режиме можно объяснить следующим образом. Точно также, как и ранее инициализируется ядро, описывается геометрия и физические процессы. Все это происходит на потоке, называемом главным или мастером, и никак не отличается от не многопоточного приложения. Затем начинается инициализация объектов классов действий и здесь начинаются отличия. Метод Build() класса G4VUserActionInitilization позволяет создавать объекты для каждого рабочего потока – потока, работа которого начинается по команде «мастер потока». Одновременно может существовать более одного потока, причем на время существования рабочих потоков мастер поток переводится в режим ожидания, а все созданные им потоки работают независимо. Когда последний рабочий поток закончит свои операции, управление программой вернется на «мастер – поток». Это означает, что во время расчета цикл обработки будет осуществляться не на одном потоке,

а на потоках, количество которых указано в объекте класса G4RunManager. По умолчанию их число равно 2.

Сразу стоит отметить, что для изменения этого значения следует

```
void SetNumberOfThreads( G4int n );
```

вызывать метод

Не плохим решением, чтобы не определять сколько потоков доступно на той или иной расчетной машине, будет использование данного метода вместе со статическим методом на определение доступного числа ядер:

вместе эта конструкция будет выглядеть следующим образом:

```
G4int G4GetNumberOfCores();
```

```
runManager->SetNumberOfThreads(G4Threading::G4GetNumberOfCores());
```

Возвращаясь к методу Build() класса G4VUserActionInitialization, при его вызове происходит генерация независимых объектов для каждого рабочего потока.

Таким образом, одновременно существует n копий генерации первичных частиц, запусков, событий и т. п. где n количество потоков.

Теперь рассмотрим, что происходит во время запуска. Если на запуск было послано 1000 событий, это совершенно не означает, что они будут равномерно распределены между потоками. На самом деле будет создана очередь событий, из которой потоки будут выхватывать следующее событие, после чего моделировать его в своей замкнутой и независимой системе. Что же касается геометрии и физических процессов, то, как известно, их нельзя изменить во время цикла обработки событий, и они доступны только для чтения рабочим потоком. Подводя итоги, можно сказать, что осуществляется максимальная производительность, однако в финале получится, что один из потоков (допустим их 2) обработал 535 событий, а другой 465.

Из-за этого возникает необходимость постобработки расчетных данных или сшивки. В оригинальном Geant4 существует класс G4SensitiveDetector или метод Merge() класса G4RunManager, однако и то и другое требует

дополнительных условностей в коде. Информацию об их использовании можно попытаться найти на официальном сайте, однако в рамках данной главы мы воспользуемся стандартным инструментом многопоточного программирования - мьютексом.

Мьютекс представляет собой одноместный семафор. Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток **блокируется** до тех пор, пока мьютекс не будет освобождён. В наших задачах мы будем использовать мьютекс самым простым образом.

Рассмотрим пример:

Пусть результат вычислений хранится в переменной `result` в классе `RunAction` наследующем `G4UserRunAction`, и пусть имеется `n` объектов класса `RunAction`, каждый из которых существует на своем рабочем потоке. Тогда необходимо сшить результаты в переменную `f_result` на мастер потоке.

Начнем с создания объекта, целью которого будет хранение суммарных данных и непосредственно сшивка. Тогда шаблон класса данного объекта будет выглядеть следующим образом:

```
class Scoring{
private:
    G4int f_result;
public:
    void setF_result(G4int f_result) {
        Scoring::f_result = f_result;
    }
}
```

Такая схема была бы верной в не многопоточном случае, однако, в данной задаче доступ на запись к переменной `f_result` будут пытаться получить все рабочие потоки, следовательно, метод `setF_result` следует защитить мьютексом.

Для этого в Geant4 уже предоставлена достаточно удобная оболочка для pthread.h.

G4MUTEX_INITIALIZER;

Сделаем мьютекс частью класса Scoring и инициализируем его.

```
class Scoring{
    G4int f_result;
    G4Mutex aMutex=G4MUTEX_INITIALIZER;
public:
    void setF_result(G4int f_result) {
        G4AutoLock l(&aMutex);
        Scoring::f_result += f_result;
    }
};
```

В свою очередь, для переключения мьютекса в закрытое состояние при вызове любым потоком метода setF_result будет служить класс G4AutoLock (на самом деле класс называется G4ImpMutexAutoLock), при вызове конструктора которого мьютекс переводится в состояние «занято», а при вызове деструктора (что происходит во время уничтожения локальной переменной l по завершению работы функции setF_result()) мьютекс переводится в состояние «открыто».

Такая структура класса Scoring защит переменную f_result от одновременного обращения двух рабочих потоков и обеспечит верность финального результата.

Что же касается объектов RunAction, то их вид будет достаточно тривиальным. Каждому из них следует передать указатель на объект класса Scoring, который будет существовать где-то под управлением мастер потока.

```

class RunAction: public G4UserRunAction{
    Scoring* scoring;
    G4int result;
    explicit RunAction(Scoring *m_scoring):scoring(m_scoring){}

public:
    void EndOfRunAction(const G4Run *aRun) override {
        scoring->setF_result(result);
    }
}

```

Такая структура объектов RunAction обеспечит автоматическую сшивку данных в финале запуска.

Объект же класса Scoring можно создать как поле потомка G4VUserActionInitialization, так как объект этого класса находится под управлением мастер потока, например,

```

class Action: public G4VUserActionInitialization{
public:
    Action::Action(){
        scoring = new Scoring();
    }

    ~Action() override{
        delete scoring;
    }

    void Build() const override{
        SetUserAction(new PrimaryPat);
        SetUserAction(new RunAction(scoring));
    }
}

```

Аналогично данному способу можно решать большинство многопоточных задач, возникающих в Geant4.

Однако в данном примере не было указано, как извлекать данные из объекта Scoring. Для решения этой задачи существует много путей, например, если математическая модель на Geant4 является частью более крупного приложения, то результат может напрямую передаваться в другую часть программы. Либо работу с объектом Scoring можно настроить с помощью пользовательских команд, о которых пойдет речь в следующей главе.

9. Встроенные команды

`/run/beamOn`

Представляет собой встроенную команду для запуска событий. Geant4 поддерживает множество встроенных команд, связанных с различными категориями. Данные команды можно использовать разными способами: из командной строки в интерактивном режиме работы Geant4, через макрос файлы или напрямую, встраивая в код разрабатываемой программы. За обработку встроенных команд отвечает класс G4UImanager.

9. Интерактивный режим

Напомним, что для работы в интерактивном режиме указатель следует получить с помощью статического метода

```
static G4UImanager * GetUImpointer();
```

```
void SessionStart();
```

и запустить непосредственно интерактивный режим за счет метода

В интерактивном режиме, во вкладке «Help» доступен весь список используемых команд.

Для большинства команд при выборе их в списке указано базовое пояснение о самой команде и её параметрах (можно ли опустить параметр, диапазон его значений, доступные кандидаты и т.д.).

9. Вызов команд из кода

Все команды в Geant4 имеют строковую форму. В Geant4 предусмотрена возможность вызова команд напрямую из кода. Для этого необходимо получить указатель на объект класса G4UImanager и воспользоваться методом

```
G4int ApplyCommand(const G4String& aCommand);
```

Данным способом можно воспользоваться из любой части кода, однако надо учитывать состояние ядра Geant4 в данный момент. К примеру, во время цикла событий нельзя вызывать команды на изменение геометрии или используемых физических процессов.

9. Макрос файлы

В Geant4 реализована поддержка внешних файлов команд. Такие файлы обычно имеют расширение *.mac. Использование этих файлов удобно, например, при настройке сцены отрисовки. Заранее можно выбрать средство отрисовки, задать необходимый угол обзора, добавить оси, логотипы и т.п., т.е. все, что приходилось бы делать вручную из команды строки при каждом запуске.

Рассмотрим некоторые особенности работы с макрос-файлами и их специальные команды.

Если существует необходимость использовать макрос-файлы в проекте, необходимо, например, вызвать метод `ApplyCommand()` для команды `/control/execute`, в качестве аргумента которой указать путь до макрос файла:

```
std::string command = "/control/execute ";
std::string fileName = argv[1];
UImanager->ApplyCommand(command + fileName);
```

где `argv` будет получен из аргументов запуска программы.

Переходя к командам, стоит начать с псевдонимов. Псевдонимы — это строковые переменные, которые можно использовать как аргументы команд. Чтобы создать псевдоним следует воспользоваться командой

```
/control/alias <aliasName> <aliasValue>
```

где `aliasName` — имя переменной `aliasValue` — значение.

Например, пусть количество вызываемых в запуске частиц передается одновременно в несколько частиц, тогда макрос-файл будет выглядеть следующим образом:

```
/control/alias numOfPat 1000
```

```
/run/beamOn {numOfPat}  
/my_com/com1 {numOfPat}  
/my_com/com2 {numOfPat}
```

Переменная псевдоним заключается в фигурные скобки. В данном случае значение 1000 будет использоваться в трех разных командах. Если значение понадобится изменить, то нужно будет изменить значение только в псевдониме.

Второй специальной командой является цикл

```
/control/loop <macroFile> <counterName> <initialValue> <finalValue> <stepSize>
```

В данном случае `macroFile` — представляет собой другой макро-файл, являющийся телом цикла

`counterName` — переменная псевдоним используемая в цикле

`initialValue` — стартовое значение псевдонима

`finalValue` — конечно значение псевдонима

`stepSize`- шаг для псевдонима

Например, пусть нужно осуществить несколько запусков подряд, причем количество частиц в запуске должно увеличиваться на 100.

Тогда `run1.mac` основной запускаемый макрос файл будет содержать следующую команду

```
/control/loop run2.mac numOfPat 100 1000 100
```

где `run2.mac` представляет собой тело цикла и содержит следующую команду:

```
/run/beamOn {numOfPat}
```

9. Создание пользовательских команд

Кроме стандартных встроенных команд пользователь может добавлять свои команды. Для этого необходимо создать объект класса, унаследованного от G4UImessenger и с помощью его методов добавить новые команды.

Класс G4UImessenger содержит виртуальный метод

```
virtual void SetNewValue(G4UIcommand * command,G4String newValue);
```

С помощью данного метода можно извлекать из команды передаваемое значение.

Рассмотрим пример по взаимодействию команды с классом действий G4VUserPrimaryGenerationAction. Для того, чтобы реализовать взаимодействие следует обеспечить связь между потомками G4VUserPrimaryGenerationAction и G4UImessenger. Учитывая этот факт, реализуем шаблон потомка G4UImessenger

```
class PrimaryPartMessenger: public G4UImessenger{
public:
    explicit PrimaryPartMessenger(PrimaryPart *);
    ~PrimaryPartMessenger() override;

    void SetNewValue(G4UIcommand*, G4String) override;
private:
    PrimaryPart* primaryPart;
};
```

В данном случае указатель primaryPart будет содержать адрес потомка G4VUserPrimaryGenerationAction.

Теперь рассмотрим непосредственно устройство команд.

Каждая команда состоит из трех частей: директории, самой команды и при необходимости передаваемого значения.



Создание команды начинается с директории. С этой целью доступен специальный класс `G4UIdirectory`. Для директории можно добавить специальное описание за счет метода

```
inline void SetGuidance(const char * aGuidance)
```

Следовательно, можно задать

директорию как:

```
primaryPartCommand = new G4UIdirectory("/PrimaryPart/");
primaryPartCommand->SetGuidance("User Primary Particles control commands");
```

Все классы команд в Geant4 наследуют `G4UIcommand`. Для всех основных типов данных используются свои потомки этого класса. К ним относятся:

- `G4UIcmdWithoutParameter`
- `G4UIcmdWithABool`
- `G4UIcmdWithAnInteger`
- `G4UIcmdWithADouble`
- `G4UIcmdWithAString`
- `G4UIcmdWith3Vector`

По умолчанию все команды принимают один параметр. Однако, если нужно добавить более одного параметра следует использовать базовый класс `G4UIcommand` за счет метода

```
inline void SetParameter(G4UIparameter *const newParameter);
```

при вызове которого в вектор параметров добавляется новый.

Теперь создадим команду, принимающую в качестве аргумента тип `G4double`

```
G4UICmdWithADouble* my_cmd;
```

Конструктор команды выглядит следующим образом:

```
G4UICmdWithADouble  
(const char * theCommandPath,G4UImessenger * theMessenger);
```

здесь theCommandPath полный путь — имя команды, theMessenger — указатель на существующий объект G4UImessenger

Соответственно инициализация команды будет выглядеть следующим образом:

```
my_cmd = new G4UICmdWithADouble("/PrimaryPart/my_cmd",this);
```

Для команд доступно несколько методов, позволяющих настраивать справочную информацию, связанную с ними, например:

```
my_cmd->SetGuidance("It's my command!");
```

добавит справочное сообщение в Help при подсветке команды.

Создать объект потомка G4UImessengerудобно в PrimaryPart как его внутренний объект:

```
class PrimaryPartMessenger;
```

```
class PrimaryPart: public G4VUserPrimaryGeneratorAction {  
private:
```

```
    G4ParticleGun *gun;  
    PrimaryPartMessenger* primaryPartMessenger;
```

```
public:
```

```
    PrimaryPart();
```

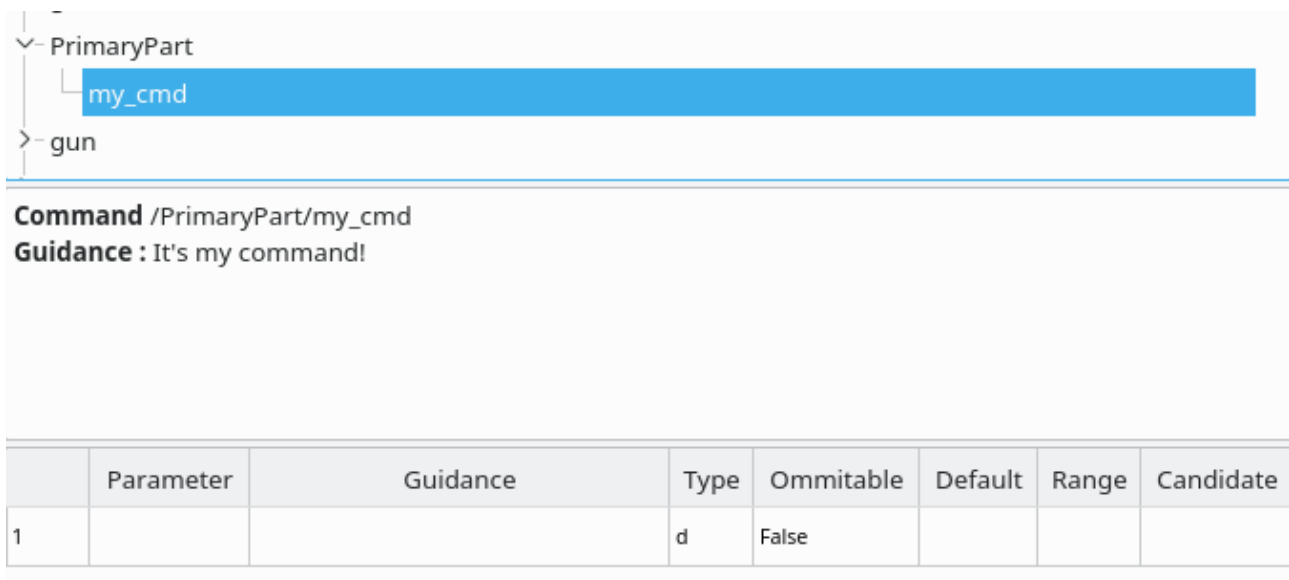
```
    ~PrimaryPart() override;
```

где инициализация объекта PrimaryPartMessenger может быть осуществлена в конструкторе

```
PrimaryPart::PrimaryPart() {
```

```
    primaryPartMessenger = new PrimaryPartMessenger(this);  
    gun = new G4ParticleGun(1);  
    gun->SetParticleDefinition(G4Neutron::NeutronDefinition());  
    gun->SetParticleEnergy(14*MeV);  
    gun->SetParticlePosition(G4ThreeVector(0,0,0));
```

Теперь, если запускать программу в интерактивном режиме, можно увидеть новую команду в списке Help.



The screenshot shows a software interface with a tree view on the left and a command details section on the right. The tree view has a root node 'PrimaryPart' which is expanded to show a sub-node 'my_cmd'. Below 'my_cmd' is another node 'gun'. The right section displays the command path 'Command /PrimaryPart/my_cmd' and its guidance 'Guidance : It's my command!'. Below this is a table with 8 columns: Parameter, Guidance, Type, Ommitable, Default, Range, and Candidate. The first row of the table has the value '1' in the first column and 'd' in the 'Type' column, with 'False' in the 'Ommitable' column.

	Parameter	Guidance	Type	Ommitable	Default	Range	Candidate
1			d	False			

Наконец, чтобы команда оказывала какое-либо влияние на класс PrimaryPart, достаточно реализовать в целевом классе соответствующий механизм, и передавать в него значение из команды в случае её вызова. Проверка команды на вызов осуществляется за счет метода SetNewValue()

```
virtual void SetNewValue(G4Uicommand * command,G4String newValue);
```

Если команда данного объекта класса была вызвана, то указатель на неё передается в качестве первого аргумента, а значение команды — в качестве второго, например:

```
void PrimaryPartMessenger::SetNewValue(G4Uicommand *command,G4String newValue) {  
    if (command == my_cmd)  
        primaryPart->foo(my_cmd->GetNewDoubleValue(newValue));  
}
```


где `void foo(G4double)` соответствующий механизм класса `PrimaryPart`. Также следует отметить, что каждый из потомков `G4Uicommand` имеет метод для конвертации своего типа данных из `G4String`.

9. Пользовательские команды, связанные с геометрией

Организация потомка `G4UImessenger`, связанного с классом описания геометрии, никак не отличается от остальных случаев. Однако команда, связанная с изменением параметров каких-либо объектов самой геометрии, требует специфических действий по её реализации. Как было сказано ранее, для инициализации геометрии вызывался метод

```
virtual G4VPhysicalVolume* Construct();
```

В рамках данного метода создаются все формы, логические объемы и физические объемы. Все эти объекты записываются в статические хранилища и продолжают существовать даже после завершения работы метода. Следовательно, для новой инициализации геометрии требуется:

- очищать все контейнеры хранилища
- вызывать метод `Construct()` вручную, т. к. автоматически он вызывается только при инициализации ядра `Geant4`.

Для очистки хранилищ достаточно следующей конструкции

```
G4GeometryManager::GetInstance()->OpenGeometry();  
G4PhysicalVolumeStore::GetInstance()->Clean();  
G4LogicalVolumeStore::GetInstance()->Clean();  
G4SolidStore::GetInstance()->Clean();
```

Её следует разместить либо в методе `Construct()`, либо внутри пользовательского механизма по изменению параметров.

После того, как все действия, ради которых вызывается команда, связанная с геометрией, выполнены, следует вызвать метод `Construct()`. Для этого требуется в конце всех операций вызвать:

```
G4RunManager::GetRunManager()->DefineWorldVolume(Construct());  
G4RunManager::GetRunManager()->ReinitializeGeometry();
```

?

10. Установка под Linux

Для установки Geant4 необходимо:

1. Подготовка

Установить пакеты *gcc*, *g++*, *make*, *qt4-dev*

Установить *cmake* версии 3.6 (или выше)

Кроме того, рекомендуется собрать пакет *Expat XML Parser* версии 2.1 или выше, однако данный при необходимости можно пропустить (см. ниже)

2. Установка Geant4

Скачать tar-архив с сайта <http://geant4.web.cern.ch/support/download> и
`tar -xf ./geant4.10.05.b01.tar.gz`

распаковать

Создать директорию для построения проекта

Перейти в созданную директорию

```
mkdir geant4.10.5-build
```

```
cd ./geant4.10.5-build/
```

```
cmake -DCMAKE_INSTALL_PREFIX=/home/idalov/geant4/geant4.10.05-install -  
DGEANT4_INSTALL_DATA=ON -DGEANT4_BUILD_MULTITHREADED=ON -DGEANT4_USE_QT=ON  
/home/idalov/geant4/geant4.10.05.b01
```

```
--DCMAKE_INSTALL_PREFIX=/path/to/geant4
```

Настроить необходимые параметры установки для *cmake*:

где

-содержит путь того, куда будет установлен Geant4одник

-DGEANT4_INSTALL_DATA=ON

-DGEANT4_USE_QT=ON

-означает что во время установки будут скачены необходимые данные сечений

-отвечает за использование qt в качестве среды для интерфейса.

-DGEANT4_BUILD_MULTITHREADED=ON

-означает что Geant4 будет установлен с поддержкой многопоточности.

Последним аргументом в команде стоит путь к исходным файлам geant4

В случае отказа от установки Expat XML Parser следует перевести следующий аргумент в состояние OFF

`-DGEANT4_USE_SYSTEM_EXPAT=OFF`

После того как все параметры настроены на экран будет выведено сообщение вида

```
-- Found Qt4: /usr/bin/qmake (found version "4.8.7")
-- Found OpenGL: /usr/lib/x86_64-linux-gnu/libOpenGL.so
-- Configuring download of missing dataset G4NDL (4.5)
-- Configuring download of missing dataset G4EMLOW (7.4)
-- Configuring download of missing dataset PhotonEvaporation (5.2)
-- Configuring download of missing dataset RadioactiveDecay (5.2)
-- Configuring download of missing dataset G4NEUTRONXS (2.0)
-- Configuring download of missing dataset G4PIL (1.3)
-- Configuring download of missing dataset RealSurface (2.1.1)
-- Configuring download of missing dataset G4SAIDDATA (1.1)
-- Configuring download of missing dataset G4ABLA (3.1)
-- Configuring download of missing dataset G4ENSDFSTATE (2.2)
-- The following Geant4 features are enabled:
GEANT4_BUILD_CXXSTD: Compiling against C++ Standard '11'
GEANT4_BUILD_MULTITHREADED: Build multithread enabled libraries
GEANT4_BUILD_TLS_MODEL: Building with TLS model 'initial-exec'
GEANT4_USE_SYSTEM_EXPAT: Using system EXPAT library
GEANT4_USE_QT: Build Geant4 with Qt support
```

Далее следует ввести команду для сборки Geant4 (где -j4 количество

`make -j4`

используемых для сборки ядер)

Эта процедура может занять продолжительное время в зависимости от возможностей ПК

Затем для окончательной сборки Geant4 следует ввести (с правами администратора):

`sudo make install`