

## 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`.

## 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.).

## 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.

## 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}
```

## 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 G4UImessnger 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 G4UImessenger удобно в PrimaryPart как его

```
class PrimaryPartMessenger;
```

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

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

```
public:
```

```
    PrimaryPart();
```

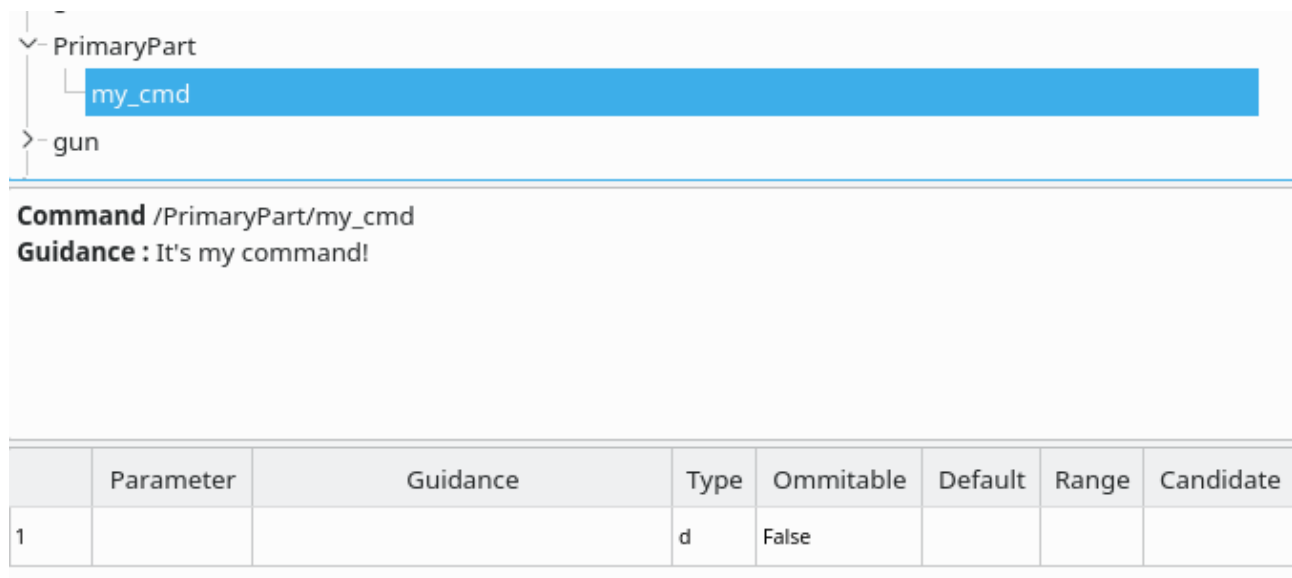
```
    ~PrimaryPart() override;
```

внутренний объект:

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 tree view on the left with 'PrimaryPart' expanded, revealing 'my\_cmd' (highlighted in blue) and 'gun'. Below the tree, the details for the selected command are shown:

**Command** /PrimaryPart/my\_cmd  
**Guidance** : It's my command!

	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`.

### **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();
```

?